CS 488/688
Winter 2006

Craig S. Kaplan, Stephen Mann

Contents

1 Administration **9**
1.1 General Information . . . 9
1.2 Topics Covered . . . 9
1.3 Assignments . . . 10

2 Introduction **11**
2.1 History . . . 11
2.2 Pipeline . . . 12
2.3 Primitives . . . 14
2.4 Algorithms . . . 14
2.5 APIs . . . 15

3 Devices and Device Independence **17**
3.1 Calligraphic and Raster Devices . . . 17
3.2 How a Monitor Works . . . 17
3.3 Physical Devices . . . 19

4 Device Interfaces **21**
4.1 Device Input Modes . . . 21
4.2 Application Structure . . . 21
4.3 Polling and Sampling . . . 22
4.4 Event Queues . . . 22
4.5 Toolkits and Callbacks . . . 23
4.6 Example for Discussion . . . 24

5 Geometries **27**
5.1 Vector Spaces . . . 27
5.2 Affine Spaces . . . 27
5.3 Euclidean Spaces . . . 28
5.4 Cartesian Space . . . 28
5.5 Summary of Geometric Spaces . . . 29

6 Affine Geometry and Transformations **31**
6.1 Linear Combinations . . . 31
6.2 Affine Combinations . . . 31
6.3 Affine Transformations . . . 32
6.4 Matrix Representation of Transformations . . . 35
6.5 Geometric Transformations . . . 36
6.6 Compositions of Transformations . . . 38
6.7 Change of Basis . . . 38
6.8 Ambiguity . . . 41
6.9 3D Transformations . . . 42
6.10 World and Viewing Frames . . . 43
6.11 Normals . . . 47

7 Windows, Viewports, NDC **49**
7.1 Window to Viewport Mapping . . . 49
7.2 Normalized Device Coordinates . . . 50

8 Clipping **53**
8.1 Clipping . . . 53

9 Projections and Projective Transformations **57**
9.1 Projections . . . 57
9.2 Why Map Z? . . . 61
9.3 Mapping Z . . . 62
9.4 3D Clipping . . . 65
9.5 Homogeneous Clipping . . . 65
9.6 Pinhole Camera vs. Camera vs. Perception . . . 67

10 Transformation Applications and Extensions **71**
10.1 Rendering Pipeline Revisited . . . 71
10.2 Derivation by Composition . . . 71
10.3 3D Rotation User Interfaces . . . 72
10.4 The Virtual Sphere . . . 73

11 Polygons **75**
11.1 Polygons – Introduction . . . 75
11.2 Polygon Clipping . . . 76
11.3 Polygon Scan Conversion . . . 79
11.4 Dos and Don'ts . . . 80

12 Hidden Surface Removal **83**
12.1 Hidden Surface Removal . . . 83
12.2 Backface Culling . . . 83
12.3 Painter's Algorithm . . . 84
12.4 Warnock's Algorithm . . . 85
12.5 Z-Buffer Algorithm . . . 86
12.6 Comparison of Algorithms . . . 87

13 Hierarchical Models and Transformations **89**
13.1 Hierarchical Transformations . . . 89
13.2 Hierarchical Data Structures . . . 92

14 Picking and 3D Selection **97**
14.1 Picking and 3D Selection . . . 97

15 Colour and the Human Visual System **101**
15.1 Colour . . . 101

16 Reflection and Light Source Models **103**
16.1 Goals . . . 103
16.2 Lambertian Reflection . . . 103
16.3 Attenuation . . . 105
16.4 Coloured Lights, Multiple Lights, and Ambient Light . . . 106
16.5 Specular Reflection . . . 107

17 Shading **111**
17.1 Introduction . . . 111
17.2 Gouraud Shading . . . 111
17.3 Phong Shading . . . 113

18 Graphics Hardware **115**
18.1 Graphics Hardware . . . 115
18.2 NVIDIA Vertex Shader Assembly . . . 117
18.3 High-Level Shading Languages . . . 119
18.4 Cg . . . 119
18.5 Sh . . . 121

19 Ray Tracing **123**
19.1 Fundamentals . . . 123
19.2 Intersection Computations . . . 124
19.3 Shading . . . 127
19.4 Recursive Ray Tracing . . . 128
19.5 Surface Information . . . 129
19.6 Modeling and CSG . . . 131
19.7 Texture Mapping . . . 133
19.8 Bounding Boxes, Spatial Subdivision . . . 136

20 Bidirectional Tracing **139**
20.1 Missing Effects . . . 139
20.2 Distribution Ray Tracing . . . 139
20.3 Bidirectional Path Tracing . . . 140
20.4 Photon Maps . . . 141

21 Aliasing **143**
21.1 Aliasing . . . 143

22 Shadows **147**
22.1 Overview . . . 147
22.2 Projective Shadows . . . 147
22.3 Shadow Maps . . . 148
22.4 Shadow Volumes . . . 150

23 Modeling of Natural Phenomena **155**
23.1 Fractal Mountains . . . 155
23.2 L-system Plants . . . 156
23.3 Particle Systems . . . 156

24 Radiosity **159**
24.1 Definitions and Overview . . . 159
24.2 Form Factors . . . 164
24.3 Progressive Refinement . . . 167
24.4 Meshing in Radiosity . . . 168

25 Photon Maps **173**
25.1 Overview . . . 173

26 Polyhedral Data Structures **177**
26.1 Storage Models . . . 177
26.2 Euler's Formula . . . 179
26.3 Winged Edge . . . 180

27 Splines **185**
27.1 Constructing Curve Segments . . . 185
27.2 Bernstein Polynomials . . . 187
27.3 Bézier Splines . . . 188
27.4 Spline Continuity . . . 191
27.5 Tensor Product Patches . . . 195
27.6 Barycentric Coordinates . . . 198
27.7 Triangular Patches . . . 200
27.8 Subdivision Surfaces . . . 202
27.9 Wavelets . . . 205

28 Non-Photorealistic Rendering **209**
28.1 2D NPR . . . 209
28.2 3D NPR . . . 213

29 Animation **217**
29.1 Overview . . . 217
29.2 Traditional 2D Cel Animation . . . 217
29.3 Automated Keyframing . . . 218
29.4 Functional Animation . . . 218
29.5 Motion Path Animation . . . 220
29.6 Orientation and Interpolation . . . 222
29.7 Quaternions . . . 224
29.8 Animating Camera Motion . . . 226
29.9 Tools for Shape Animation . . . 227
29.10Kinematics and Inverse Kinematics . . . 229
29.11Physically-Based Animation . . . 230
29.12Human Motion . . . 232
29.13Sensor-Actuator Networks . . . 233
29.14Morphing . . . 234
29.15Motion Capture . . . 235
29.16Flocking . . . 237

30 Assignments **239**
30.1 Assignment 0: Introduction 239
30.2 Assignment 1: Introduction to OpenGL 239
30.3 Assignment 2: Frames and Perspective 240
30.4 Assignment 3: Hierarchical Modelling 241
30.5 Assignment 4: A Raytracer 242
30.6 Assignment 5: The Project 243

These notes contain the material that appears on the overhead slides (or on the computer) shown in class. You should, however, read the course text and attend lectures to fill in the missing details. The material in this course is organized around the assignments, with the material at the end of the course being a sequence of topics not covered on the assignments. Some of the images shown on the slides cannot be included in the printed course notes for copyright reasons.

The following former CS 488/688 students have images appearing in these notes:

- Eric Hall
- Ian McIntyre
- Selina Siu
- Bryan Chan
- Franke Belme
- Andrew Lau
- Jerome Carriére
- Bonnie Liu
- Zaid Mian
- Ryan Meridith-Jones

1 Administration

1.1 General Information

Instructors: Craig S. Kaplan, Stephen Mann

Electronic Resources:
Web: `http://www.student.cs.uwaterloo.ca/~cs488/`
Newsgroup: `uw.cs.cs488`
Email: `cs488@cgl.uwaterloo.ca`

When sending email from non-uwaterloo account,
put "CG:" in the subject

Texts:

- D. Hearn and M.P. Baker,
  Computer Graphics, 3rd ed. (required)
- Course Overview and Notes (online or at Bookstore)
- *OpenGL Programming Guide* (optional)

Assignments:
5 Assignments (A0-A4) plus a Project (A0 is worth 0 marks).

Marking Algorithm:
Assignments: 32%; Project: 24%; Midterm: 14%; Final: 30%.
Must obtain 50% in both Programming and Exam portions to pass.

Programming Environment:
C++, OpenGL, Lua; Linux PCs.

General Comments:
A tough course: lots of work! Be committed.
If you don't know C++, you will have major problems.
Do **NOT** take graphics and either real-time (CS452) or compilers (CS444).

1.2 Topics Covered

- Graphics Pipeline and Hardware
- Mathematical Foundations:
  - Affine and Projective Geometry, Linear Algebra and Transformations
  - Numerical Analysis and Splines
- Modelling and Data Structures
- Hidden Surface Removal
- Colour and the Human Visual System

- Lighting and Shading
- Ray Tracing
- Global Illumination (optional; if time permits)
- Animation (optional; if time permits)

1.3 Assignments

Number: 5 Assignments (A0-A4) plus a Project (A0 worth 0 marks)

Environment: Linux PCs.
You are expected to share machines. Be nice!

Code Credit: ***You must request in*** *`README`*.
For each missed objective, in `README` state what is wrong and how you would attempt to fix it, mark up printout of code.

Hand in: At start of class (first 5 minutes). NO LATES!
Hand in documentation only; code read online.

Documentation Requirements:

- Title page (with name, student ID, userid)
- ***Signed*** *objective list:* 10 points you are graded on; see course notes.
- `README` and Manual
- Annotated hardcopy (only if code broken)
- Checksum (run `/u/gr/cs488/bin/grsubmit` to obtain)

Assignment Format:

- *Very strict!*
- Read assignment policies in course notes.
- **Do A0 to avoid losing marks later.**
- See (and use) checklist.

Code:

- Run `/u/gr/cs488/bin/setup`.
- All files need to be `cs488` group owned and group readable.
- See course notes for assignment specs and objectives.

2 Introduction

2.1 History

A Brief History of Computer Graphics

Early 60s: Computer animations for physical simulation;
Edward Zajac displays satellite research using CG in 1961

1963: Sutherland (MIT)
Sketchpad (direct manipulation, CAD)

Calligraphic (vector) display devices
Interactive techniques
Douglas Englebart invents the mouse.

1968: Evans & Sutherland founded

1969: First SIGGRAPH

Late 60's to late 70's: Utah Dynasty

1970: Pierre Bézier develops Bézier curves

1971: Gouraud Shading

1972: Pong developed

1973: Westworld, The first film to use computer animation

1974: Ed Catmull develops z-buffer (Utah)
First Computer Animated Short, *Hunger*:
Keyframe animation and morphing

1975: Bui-Tuong Phong creates Phong Shading (Utah)
Martin Newell models a teapot with Bézier patches (Utah)

Mid 70's: Raster graphics (Xerox PARC, Shoup)

1976: Jim Blinn develops texture and bump mapping

1977: Star Wars, CG used for Death Star plans

1979: Turner Whitted develops ray tracing

Mid 70's - 80's: Quest for realism
radiosity; also mainstream real-time applications.

1982: Tron, Wrath of Khan. Particle systems and obvious CG

1984: The Last Starfighter, CG replaces physical models. Early attempts at realism using CG

1986: First CG animation nominated for an Academy Award: Luxo Jr. (Pixar)

1989: Tin Toy (Pixar) wins Academy Award

1995: Toy Story (Pixar and Disney), the first full length fully computer-generated 3D animation
Reboot, the first fully 3D CG Saturday morning cartoon
Babylon 5, the first TV show to routinely use CG models

late 90's: Interactive environments, scientific and medical visualization, artistic rendering, image based rendering, path tracing, photon maps, etc.

00's: Real-time photorealistic rendering on consumer hardware? Interactively rendered movies?

(Readings: Watt, Preface (optional), Hearn and Baker, Chapter 1 (optional), Red book [Foley, van Dam et al: Introduction to Computer Graphics], Chapter 1 (optional). White book [Foley, van Dam et al: Computer Graphics: Principles and Practice, 2nd ed.], Chapter 1 (optional).)

2.2 Pipeline

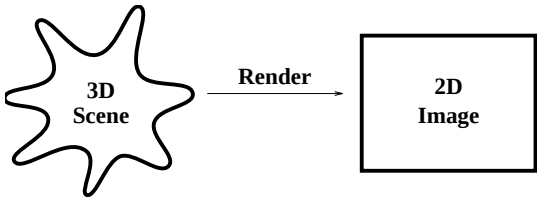
We begin with a description of **forward rendering**, which is the kind of rendering usually supported in hardware and is the model OpenGL uses. In forward rendering, rendering primitives are transformed, usually in a conceptual pipeline, from the model to the device.

However, **raytracing**, which we will consider later in the course, is a form of **backward rendering**. In backward rendering, we start with a point in the image and work out what model primitives project to it.

Both approaches have advantages and disadvantages.

The Graphics Pipeline

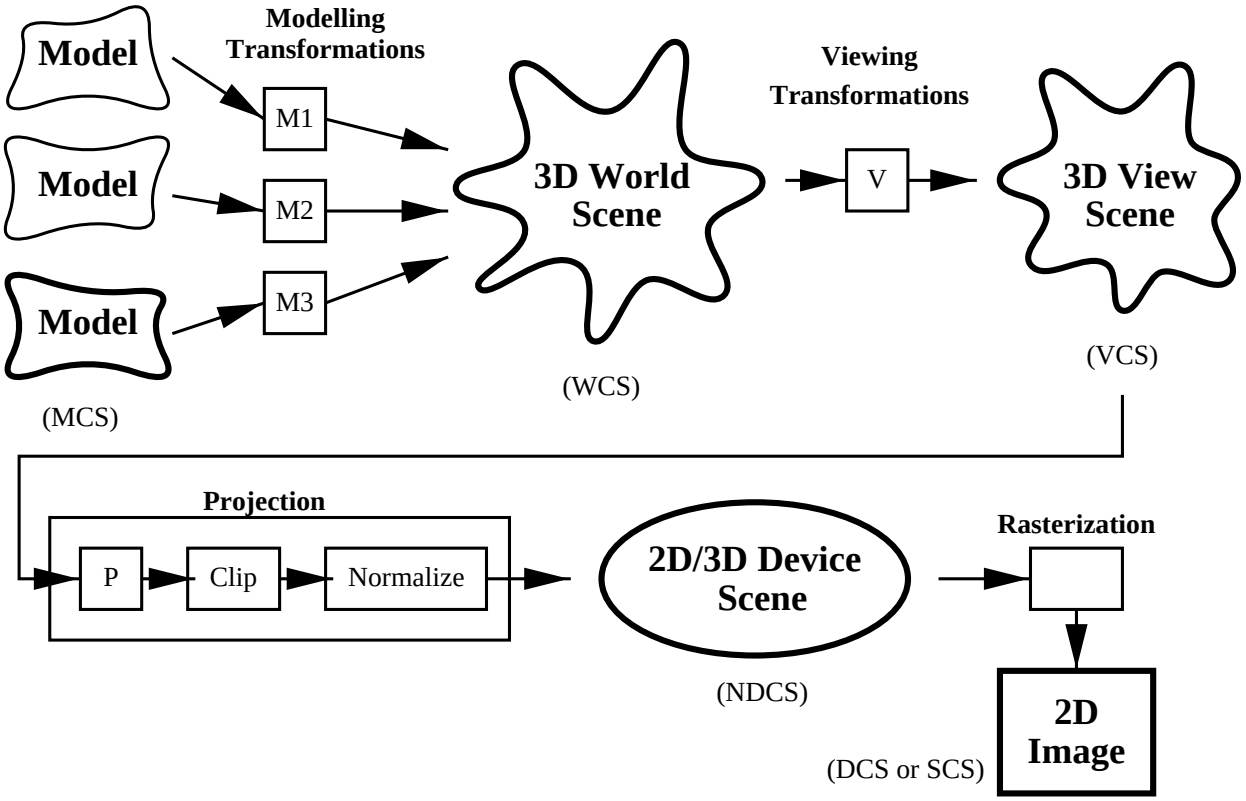
- **Rendering** is the conversion of a **scene** into an **image**:



- Scenes are composed of **models** in three-dimensional space.
  Models are composed of **primitives** supported by the rendering system.
- Models entered by hand or created by a program.
- The image is drawn on monitor, printed on laser printer, or written to a raster in memory or a file.
  Requires us to consider **device independence**.
- Classically, "model" to "scene" to "image" conversion broken into finer steps, called the **graphics pipeline**.
- Parts of pipeline implemented in graphics hardware to get interactive speeds.
- Modern hardware acceleration supports *programmable* pipelines.
  - Some of pipeline stages are programmable units

  - You can implement your own lighting models or geometric transformations.
  - Programmable graphics accelerator: GPU
  - You have to know what you're doing (i.e. know the math) to use a GPU!

- The basic **forward projection** pipeline looks like:



- Each stage refines the scene,
  converting primitives in modelling space
  to primitives in device space,
  where they are converted to pixels (rasterized).

- A number of coordinate systems are used:

  MCS: Modelling Coordinate System.

  WCS: World Coordinate System.

  VCS: Viewer Coordinate System.

  NDCS: Normalized Device Coordinate System.

  DCS or SCS: Device Coordinate System or equivalently the Screen Coordinate System.

  Keeping these straight is the key to understanding a rendering system.

- Transformation between two coordinate systems represented with matrix.

- Derived information may be added (lighting and shading) and primitives may be removed (hidden surface removal) or modified (clipping).

(Readings: Readings: Watt, Chapter 5 introduction, 5.1. Hearn and Baker, Section 6-1 (but they give a more detailed version than used here). Red book, 6-1, 6-6 (intro). White book, 8-3, Blinn: 16.)

2.3 Primitives

- Models are composed of/converted to **geometric primitives**.
- Typical rendering primitives directly supported in hardware:
  - Points (single pixels)
  - Line Segments
  - Polygons (perhaps only convex polygons or triangles).
- Modelling primitives also include
  - Piecewise polynomial (spline) curves
  - Piecewise polynomial (spline) surfaces
  - Implicit surfaces (quadrics, blobbies, etc)
  - Other...
- Software renderer may support modelling primitives directly, or may convert them into polygonal or linear approximations for hardware rendering.

2.4 Algorithms

A number of basic algorithms are needed:

- **Transformation:** Convert representations of models/primitives from one coordinate system to another.
- **Clipping/Hidden Surface Removal:** Remove primitives and parts of primitives that are not visible on the display.
- **Rasterization:** Convert a projected screen-space primitive to a set of pixels.

Later, we will look at some more advanced algorithms:

- **Picking:** Select a 3D object by clicking an input device over a pixel location.
- **Shading and Illumination:** Simulate the interaction of light with a scene.
- **Animation:** Simulate movement by rendering a sequence of frames.

2.5 APIs

Application Programming Interfaces (APIs):

Xlib, GDI: 2D rasterization.

PostScript, PDF, SVG: 2D transformations, 2D rasterization

OpenGL, Direct3D: 3D pipeline

APIs provide access to rendering hardware via a conceptual model.

APIs hide which graphics algorithms are or are not implemented in hardware by simulating missing pieces in software.

For interactive applications, we might modify the scene or a model directly or just the viewing information, then regenerate the image.

3 Devices and Device Independence

3.1 Calligraphic and Raster Devices

Calligraphics and Raster Devices

Calligraphic Display Devices draw polygon and line segments directly:

- Plotters
- Direct Beam Control CRTs
- Laser Light Projection Systems

Raster Display Devices represent an image as a regular grid of **samples**.

- Each sample is usually called a **pixel** ("picture element")
- Rendering requires **rasterization algorithms** to quickly determine a sampled representation of geometric primitives.

3.2 How a Monitor Works

How a CRT Works

Raster Cathode Ray Tubes (CRTs) most common display device

- Capable of high resolution.
- Good colour fidelity.
- High contrast (100:1).
- High update rates.

Electron beam scanned in regular pattern of horizontal **scanlines**.

Raster images stored in a **frame buffer**.

Frame buffers composed of **VRAM** (video RAM).

VRAM is dual-ported memory capable of

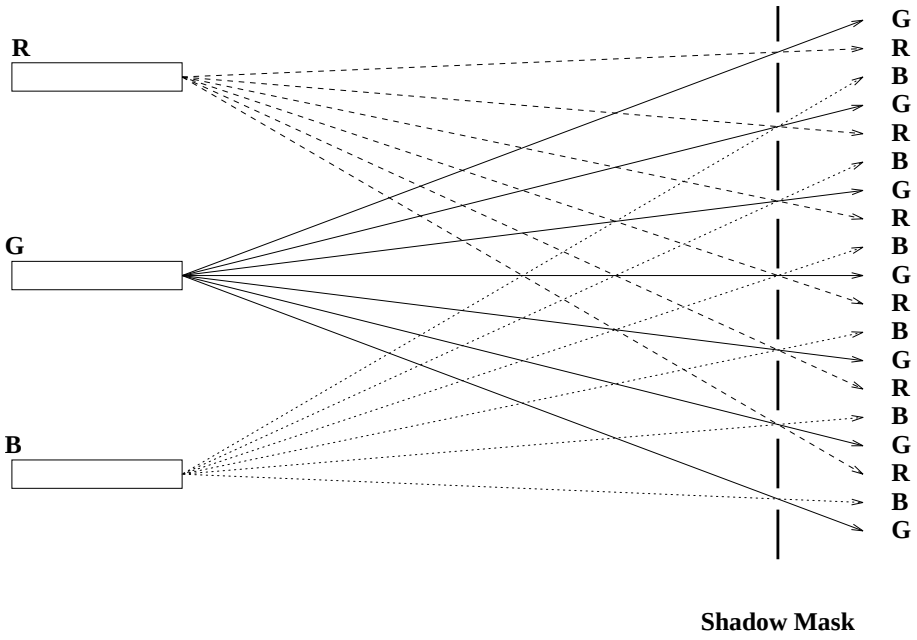
- Random access
- Simultaneous high-speed serial output: built-in **serial shift register** can output entire *scanline* at high rate synchronized to **pixel clock.**

Intensity of electron beam modified by the pixel value.

Burst-mode DRAM replacing VRAM in many systems.

Colour CRTs have three different colours of phosphor and three independent electron guns.

Shadow Masks allow each gun to irradiate only one colour of phosphor.



Colour is specified either

- Directly, using three independent intensity channels, or
- Indirectly, using a **Colour Lookup Table (LUT)**.

In the latter case, a **colour index** is stored in the frame buffer.

Sophisticated frame buffers may allow different colour specifications for different portions of frame buffer. Use a **window identifier** also stored in the frame buffer.

How Liquid Crystal Displays Work

Liquid Crystal Displays (LCDs)

- Flat panels
- Flicker free
- Decreased viewing angle

Works as follows:

- Random access to cells like memory.
- Cells contain liquid crystal molecules that align when charged.
- Unaligned molecules twist light.
- Polarizing filters allow only light through unaligned molecules.
- Subpixel colour filter masks used for RGB.

(Readings: Watt, none. Hearn and Baker, Chapter 2. Red book, Chapter 4. White book, Chapter 4.

LCD reference: http://www.cgl.uwaterloo.ca/~pogilhul/present/lcd)

3.3 Physical Devices

Physical Devices

Physical input devices include

- Dials (Potentiometers)
- Sliders
- Pushbuttons
- Switches
- Keyboards (collections of pushbuttons called "keys")
- Trackballs (relative motion)
- Mice (relative motion)
- Joysticks (relative motion, direction)
- Tablets (absolute position)
- Etc.

Need some abstractions to keep organized...

(Readings: Watt: none. Hearn and Baker, Chapter 8. Red book: 8.1. White book: 8.1.)

4 Device Interfaces

4.1 Device Input Modes

Device Input Modes

Input from devices may be managed in different ways:

Request Mode: Alternating application and device execution

- application requests input and then suspends execution;
- device wakes up, provides input and then suspends execution;
- application resumes execution, processes input.

Sample Mode: Concurrent application and device execution

- device continually updates register(s)/memory location(s);
- application may read at any time.

Event Mode: Concurrent application and device execution together with a concurrent queue management service

- device continually offers input to the queue
- application may request selections and services from the queue
  (or the queue may interrupt the application).

4.2 Application Structure

Application Structure

With respect to device input modes, applications may be structured to engage in

- **requesting**
- **polling** or **sampling**
- **event processing**

Events may or may not be **interruptive**.
If not interruptive, they may be read in a

- **blocking**
- **non-blocking**

fashion.

4.3 Polling and Sampling

Polling

In **polling**,

- Value of input device constantly checked in a tight loop
- Wait for a change in status

Generally, polling is inefficient and should be avoided, particularly in time-sharing systems.

Sampling

In **sampling**, value of an input device is read and then the program proceeds.

- No tight loop
- Typically used to track sequence of actions (the mouse)

4.4 Event Queues

Event Queues

- Device is monitored by an asynchronous process.
- Upon change in status of device, this process places a record into an **event queue.**
- Application can request read-out of queue:
  - Number of events
  - First waiting event
  - Highest priority event
  - First event of some category
  - All events
- Application can also
  - Specify which events should be placed in queue
  - Clear and reset the queue
  - Etc.
- Queue reading may be blocking or non-blocking
- Processing may be through callbacks
- Events may be processed interruptively
- Events can be associated with more than physical devices...
  Windowing system can also generate virtual events, like "Expose".

Without interrupts, the application will engage in an *event loop*

- not a tight loop
- a preliminary of *register event* actions followed by a repetition of *test for event* actions.

For more sophisticated queue management,

- application merely registers event-process pairs
- queue manager does all the rest
  "if event E then invoke process P."
- Events can be restricted to particular areas of the screen, based on the cursor position.
- Events can be very general or specific:
  - A mouse button or keyboard key is depressed.
  - The cursor enters a window.
  - The cursor has moved more than a certain amount.
  - An `Expose` event is triggered under X when a window becomes visible.
  - A `Configure` event is triggered when a window is resized.
  - A timer event may occur after a certain interval.
- Simple event queues just record a code for event (Iris GL).
- Better event queues record extra information such as time stamps (X windows).

4.5 Toolkits and Callbacks

Toolkits and Callbacks

Event-loop processing can be generalized:

- Instead of `switch`, use table lookup.
- Each table entry associates an event with a **callback** function.
- When event occurs, corresponding **callback** is invoked.
- Provide an API to make and delete table entries.
- Divide screen into parcels, and assign different callbacks to different parcels (X Windows does this).
- Event manager does most or all of the administration.

Modular UI functionality is provided through a set of **widgets**:

- **Widgets** are parcels of the screen that can respond to events.

  Graphical representation that suggests function.

- Widgets may respond to events with a change in appearance, as well as issuing callbacks.
- Widgets are arranged in a parent/child hierarchy.
  - Event-process definition for parent may apply to child, and child may add additional event-process definitions
  - Event-process definition for parent may be redefined within child
- Widgets may have multiple parts, and in fact may be composed of other widgets in a hierarchy.

Some UI toolkits:

- Tk
- Tkinter
- Motif
- Gtk
- Qt
- WxWindows
- FLTK
- AWT
- Swing
- XUL
- ...

UI toolkits recommended for projects: Gtkmm, GLUT, SDL.

4.6 Example for Discussion

Example for Discussion

```
#include <gtkmm.h>
#include <iostream>

void test() { std::cout << "Hello, World!" << std::endl; }
int main( int argc, char *argv[] ) {
  Gtk::Main kit( argc, argv );
  Gtk::Window hw;
  hw.set_border_width( 10 );

  Gtk::Button b( "Click me" );
```

```
  b.signal_clicked().connect( &test );
  hw.add( b );
  b.show();

  Gtk::Main::run( hw );
}
```

1. Where is the definition of the event?
2. What constitutes the event?
3. What process is associated with the event?
4. Who manages the event loop?

5 Geometries

5.1 Vector Spaces

Vector Spaces

Definition:

- Set of vectors $\mathcal{V}$.
- Two operations. For $\vec{v}, \vec{u} \in \mathcal{V}$:
  Addition: $\vec{u} + \vec{v} \in \mathcal{V}$.
  Scalar multiplication: $\alpha\vec{u} \in \mathcal{V}$, where α is a member of some field $\mathbb{F}$, (i.e. $\mathbb{R}$).

Axioms:

Addition Commutes: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$.
Addition Associates: $(\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$.
Scalar Multiplication Distributes: $\alpha(\vec{u} + \vec{v}) = \alpha\vec{u} + \alpha\vec{v}$.
Unique Zero Element: $\vec{0} + \vec{u} = \vec{u}$.
Field Unit Element: $1\vec{u} = \vec{u}$.

Span:

- Suppose $\mathcal{B} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$.
- $\mathcal{B}$ **spans** $\mathcal{V}$ iff any $\vec{v} \in \mathcal{V}$ can be written as $\vec{v} = \sum_{i=1}^{n} \alpha_i \vec{v}_i$.
- $\sum_{i=1}^{n} \alpha_i \vec{v}_i$ is a **linear combination** of the vectors in $\mathcal{B}$.

Basis:

- Any minimal spanning set is a basis.
- All bases are the same size.

Dimension:

- The number of vectors in any basis.
- We will work in 2 and 3 dimensional spaces.

5.2 Affine Spaces

Affine Space

Definition: Set of Vectors $\mathcal{V}$ and a Set of Points $\mathcal{P}$

- Vectors $\mathcal{V}$ form a **vector space**.
- Points can be combined with vectors to make new points:
  $P + \vec{v} \Rightarrow Q$ with $P, Q \in \mathcal{P}$ and $\vec{v} \in \mathcal{V}$.

Frame: An affine extension of a basis:
Requires a vector basis plus a point $\mathcal{O}$ (the **origin**):

$$F = (\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n, \mathcal{O})$$

Dimension: The dimension of an affine space is the same as that of $\mathcal{V}$.

5.3 Euclidean Spaces

Euclidean Spaces

Metric Space: Any space with a **distance metric** $d(P, Q)$ defined on its elements.

Distance Metric:

- Metric $d(P, Q)$ must satisfy the following axioms:
  1. $d(P, Q) \geq 0$
  2. $d(P, Q) = 0$ iff $P = Q$.
  3. $d(P, Q) = d(Q, P)$.
  4. $d(P, Q) \leq d(P, R) + d(R, Q)$.
- Distance is intrinsic to the space, and *not* a property of the frame.

Euclidean Space: Metric is based on a dot (inner) product:

$$d^2(P, Q) = (P - Q) \cdot (P - Q)$$

Dot product:

$$\begin{aligned} (\vec{u} + \vec{v}) \cdot \vec{w} &= \vec{u} \cdot \vec{w} + \vec{v} \cdot \vec{w}, \\ \alpha(\vec{u} \cdot \vec{v}) &= (\alpha\vec{u}) \cdot \vec{v} \\ &= \vec{u} \cdot (\alpha\vec{v}) \\ \vec{u} \cdot \vec{v} &= \vec{v} \cdot \vec{u}. \end{aligned}$$

Norm:

$$|\vec{u}| = \sqrt{\vec{u} \cdot \vec{u}}.$$

Angles:

$$\cos(\angle\vec{u}\vec{v}) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}$$

Perpendicularity:

$$\vec{u} \cdot \vec{v} = 0 \ \Rightarrow \vec{u} \perp \vec{v}$$

Perpendicularity is *not* an affine concept! There is no notion of angles in affine space.

5.4 Cartesian Space

Cartesian Space

Cartesian Space: A Euclidean space with an standard **orthonormal frame** $(\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O})$.

Orthogonal: $\vec{\imath} \cdot \vec{\jmath} = \vec{\jmath} \cdot \vec{k} = \vec{k} \cdot \vec{\imath} = 0$.

Normal: $|\vec{\imath}| = |\vec{\jmath}| = |\vec{k}| = 1$.

Notation: Specify the **Standard Frame** as $F_S = (\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O})$.

As defined previously, points and vectors are

- Different objects.
- Have different operations.
- Behave differently under transformation.

Coordinates: Use an "an extra coordinate":

- 0 for vectors: $\vec{v} = (v_x, v_y, v_z, 0)$ means $\vec{v} = v_x\vec{\imath} + v_y\vec{\jmath} + v_z\vec{k}$.
- 1 for points: $P = (p_x, p_y, p_z, 1)$ means $P = p_x\vec{\imath} + p_y\vec{\jmath} + p_z\vec{k} + \mathcal{O}$.
- Later we'll see other ways to view the fourth coordinate.
- **Coordinates have no meaning without an associated frame!**
  - Sometimes we'll omit the extra coordinate ...
    point or vector by context.
  - Sometimes we may not state the frame ...
    the Standard Frame is assumed.

5.5 Summary of Geometric Spaces

Space	**Objects**	**Operators**	
Vector Space	Vector	$\vec{u} + \vec{v}$, $\alpha\vec{v}$	
Affine Space	Vector, Point	$\vec{u} + \vec{v}$, $\alpha\vec{v}$, $P + \vec{v}$	
Euclidean Space	Vector Point	$\vec{u} + \vec{v}$, $\alpha\vec{v}$, $P + \vec{v}$, $\vec{u} \cdot \vec{v}$, $d(P, Q)$	Question: where does the cross
Cartesian Space	Vector, Point, O.N. Frame	$\vec{u} + \vec{v}$, $\alpha\vec{v}$, $P + \vec{v}$, $\vec{u} \cdot \vec{v}$, $d(P, Q)$, $\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O}$	product enter?

(Readings: Watt: 1.1, 1.2. White book: Appendix A. Hearn and Baker: A-2, A-3.)

6 Affine Geometry and Transformations

6.1 Linear Combinations

Linear Combinations

Vectors:

$$\mathcal{V} = \{\vec{u}\}, \quad \vec{u} + \vec{v}, \quad \alpha\vec{u}$$

By Extension:

$$\sum_i \alpha_i \vec{u}_i \ \text{ (no restriction on } \alpha_i)$$

Linear Transformations:

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}), \quad T(\alpha\vec{u}) = \alpha T(\vec{u})$$

By Extension:

$$T(\sum_i \alpha_i \vec{u}_i) = \sum_i \alpha_i T(\vec{u}_i)$$

Points:

$$\mathcal{P} = \{P\}, \quad P + \vec{u} = Q$$

By Extension...

6.2 Affine Combinations

Affine Combinations

Define Point Subtraction:

$Q - P$ means $\vec{v} \in \mathcal{V}$ such that $Q = P + \vec{v}$ for $P, Q \in \mathcal{P}$.

By Extension:

$$\sum \alpha_i P_i \text{ is a } \textbf{vector} \text{ iff } \sum \alpha_i = 0$$

Define Point Blending:

$Q = (1 - a)Q_1 + aQ_2$ means $Q = Q_1 + a(Q_2 - Q_1)$ with $Q \in \mathcal{P}$

Alternatively:

we may write $Q = a_1 Q_1 + a_2 Q_2$ where $a_1 + a_2 = 1$.

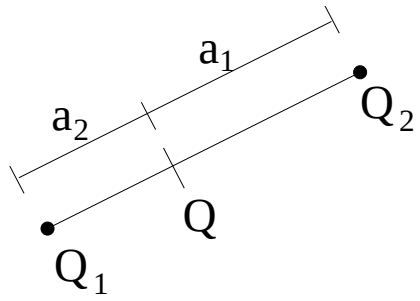
By Extension:

$$\sum a_i P_i \text{ is a } \textbf{point} \text{ iff } \sum a_i = 1$$

Geometrically:

- The following ratio holds for $Q = a_1 Q_1 + a_2 Q_2$

$$\frac{|Q - Q_1|}{|Q - Q_2|} = \frac{a_2}{a_1} \quad (a_1 + a_2 = 1)$$



- If Q breaks the line segment $\overline{Q_1Q_2}$ into the ratio $b_2 : b_1$ then

$$Q = \frac{b_1Q_1 + b_2Q_2}{b_1 + b_2} \quad (b_1 + b_2 \neq 0)$$

Legal vector combinations:
Vectors can be formed into any combinations $\sum_i \alpha_i \vec{u}_i$ (a "linear combination").

Legal point combinations:
Points can be formed into combinations $\sum_i a_i P_i$ iff

- The coefficients sum to 1: The result is a point (an "affine combination").
- The coefficients sum to 0: The result is a vector (a "vector combination").

Parametric Line Equation:
has geometric meaning (in an affine sense):

$$\begin{aligned} L(t) &= A + t(B - A) \\ &= (1 - t)A + tB \end{aligned}$$

The weights t and $(1 - t)$ create an affine combination.
The result is a point (on the line).

Parametric Ray Equation:
Same as above, but $t \geq 0$. Will write as

$$R(t) = A + t\vec{d}$$

Where A is the point of origin and $\vec{d}$ is the direction of the ray. Used in ray-tracing.

(Readings: White book, Appendix A)

6.3 Affine Transformations

Affine Transformations

Let $T : \mathcal{A}_1 \longmapsto \mathcal{A}_2$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ are affine spaces.

Then T is said to be an affine transformation if:

- T maps vectors to vectors and points to points

- T is a linear transformation on the vectors
- $T(P + \vec{u}) = T(P) + T(\vec{u})$

By Extension:

T preserves affine combinations on the points:

$$T(a_1Q_1 + \cdots + a_nQ_n) = a_1T(Q_1) + \cdots + a_nT(Q_n),$$

If $\sum a_i = 1$, result is a point.
If $\sum a_i = 0$, result is a vector.

Observations:

- Affine transformations map lines to lines:

$$T((1-t)P_0 + tP_1) = (1-t)T(P_0) + tT(P_1)$$

- Affine transformations map rays to rays:

$$T(A + t\vec{d}) = T(A) + tT(\vec{d})$$

- Affine transformations preserve ratios of distance along a line (converse is also true: preserves ratios of such distances $\Rightarrow$ affine).
- Absolute distances or angles may not be preserved.
  Absolute distances and angles are not affine concepts...

Examples:

- translations
- rotations
- scales
- shears
- reflections

Translations and rotations called *rigid body motions*.

Affine vs Linear

Which is a larger class of transformations: Affine or Linear?

- All affine transformations are linear transformations on vectors.
- Consider identity transformation on vectors.

  There is one such linear transformation.
- Consider Translations:
  - Identity transformation on vectors
  - Infinite number different ones (based on effect on points)

  Thus, there are an infinite number of affine transformations that are identity transformation on vectors.
- What makes affine transformations a bigger class than linear transformation is their translational behaviour on points.

Extending Affine Transformations to Vectors

Suppose we only have T defined on points.

Define $T(\vec{v})$ as follows:

- There exist points Q and R such that $\vec{v} = Q - R$.
- Define $T(\vec{v})$ to be $T(Q) - T(R)$.

Note that Q and R are not unique.

The definition works for $P + \vec{v}$:

$$\begin{aligned} T(P + \vec{v}) &= T(P + Q - R) \\ &= T(P) + T(Q) - T(R) \\ &= T(P) + T(\vec{v}) \end{aligned}$$

Can now show that the definition is well defined.

Theorem: Affine transformations map parallel lines to parallel lines.

Mapping Through an Affine Transformation

Need to pick a numerical representation; use *coordinates*:

Let $\mathcal{A}$ and $\mathcal{B}$ be affine spaces.

- Let $T : \mathcal{A} \mapsto \mathcal{B}$ be an affine transformation.
- Let $F_{\mathcal{A}} = (\vec{v}_1, \vec{v}_2, \mathcal{O}_{\mathcal{V}})$ be a frame for $\mathcal{A}$.
- Let $F_{\mathcal{B}} = (\vec{w}_1, \vec{w}_2, \mathcal{O}_{\mathcal{W}})$ be a frame for $\mathcal{B}$.
- Let P be a point in $\mathcal{A}$ whose *coordinates* relative $F_{\mathcal{A}}$ are $(p_1, p_2, 1)$.
  $(P = p_1\vec{v}_1 + p_2\vec{v}_2 + 1\mathcal{O}_{\mathcal{V}})$

$\mathcal{O}_{\mathcal{V}}$ and $\mathcal{O}_{\mathcal{W}}$ are called the *origins* of their respective frames.

Question: What are the coordinates $(p'_1, p'_2, 1)$ of $T(P)$ relative to the frame $F_{\mathcal{B}}$?

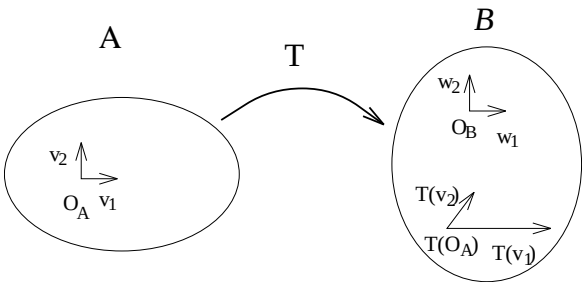
Fact: An affine transformation is completely characterized by the image of a frame in the domain:

$$\begin{aligned} T(P) &= T(p_1\vec{v}_1 + p_2\vec{v}_2 + \mathcal{O}_{\mathcal{V}}) \\ &= p_1T(\vec{v}_1) + p_2T(\vec{v}_2) + T(\mathcal{O}_{\mathcal{V}}). \end{aligned}$$

If

$$\begin{aligned} T(\vec{v}_1) &= t_{1,1}\vec{w}_1 + t_{2,1}\vec{w}_2 \\ T(\vec{v}_2) &= t_{1,2}\vec{w}_1 + t_{2,2}\vec{w}_2 \\ T(\mathcal{O}_{\mathcal{V}}) &= t_{1,3}\vec{w}_1 + t_{2,3}\vec{w}_2 + \mathcal{O}_{\mathcal{W}} \end{aligned}$$

then we can find $(p'_1, p'_2, 1)$ by substitution and gathering like terms.



(Readings: White book, Appendix A)

6.4 Matrix Representation of Transformations

Matrix Representation of Transformations

Represent points and vectors as $n \times 1$ matrices. In 2D,

$$P \equiv \mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix}$$

$$\vec{v} \equiv \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ 0 \end{bmatrix}$$

This is a Shorthand: Coordinates are specified relative to a frame $F = (\vec{v}_1, \vec{v}_2, \mathcal{O}_\mathcal{V})$:

$$\begin{aligned} P &\equiv [\vec{v}_1, \vec{v}_2, \mathcal{O}_\mathcal{V}] \begin{bmatrix} p_1 \\ p_2 \\ 1 \end{bmatrix} \\ &= F\mathbf{p}. \end{aligned}$$

Technically,

- Should write the $\mathbf{p}$ as $\mathbf{p}_F$.
- The frame F should note what space it's in.

Usually, we're lazy and let '$\mathbf{p}$' denote both

- The point
- Its matrix representation relative to an understood frame.

Transformations:

- $F_A = (\vec{v}_1, \vec{v}_2, \mathcal{O}_\mathcal{V})$ is the frame of the domain,
- $F_B = (\vec{w}_1, \vec{w}_2, \mathcal{O}_\mathcal{W})$ is the frame of the range.

Then ...

$P = \mathbf{p}$ transforms to $\mathbf{p}' = M_T\mathbf{p}$.

Can also read this as $F_A = F_B M_T$.

M_T is said to be the **matrix representation** of T relative to F_A and F_B.

- First column of M_T is representation of $T(\vec{v}_1)$ in F_B.
- Second column of M_T is representation of $T(\vec{v}_2)$ in F_B.
- Third column of M_T is representation of $T(\mathcal{O}_\mathcal{V})$ in F_B.

6.5 Geometric Transformations

Geometric Transformations

Construct matrices for simple geometric transformations.

Combine simple transformations into more complex ones.

- Assume that the range and domain frames are the Standard Frame.
- Will begin with 2D, generalize later.

Translation: Specified by the vector $[\Delta x, \Delta y, 0]^T$:

- A point $[x, y, 1]^T$ will map to $[x + \Delta x, y + \Delta y, 1]^T$.
- A vector will remain unchanged under translation.
- Translation is **NOT** a linear transformation.
- Translation is linear on sums of vectors...

Matrix representation of translation

- We can create a matrix representation of translation:

$$\overbrace{\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}}^{T(\Delta x, \Delta y)} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{bmatrix}$$

- $T(\Delta x, \Delta y)$ will mean the above matrix.
- Note that vectors are unchanged by this matrix.
- Although more expensive to compute than the other version of translation, we prefer this one:
  - Uniform treatment of points and vectors
  - Other transformations will also be in matrix form.
    We can compose transformations by matrix multiply. Thus, the composite operation less expensive if translation composed, too.

Scale about the origin:
Specified by factors $s_x, s_y \in \mathbb{R}$.

- Applies to points or vectors, is linear.

- A point $[x, y, 1]^T$ will map to $[s_x x, s_y y, 1]^T$.
- A vector $[x, y, 0]^T$ will map to $[s_x x, s_y y, 0]^T$.
- Matrix representation:

$$\overbrace{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{S(s_x,s_y)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ 0 \text{ or } 1 \end{bmatrix}$$

Rotation: Counterclockwise about the origin, by angle θ.

- Applies to points or vectors, is linear.
- Matrix representation:

$$\overbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{R(\theta)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 0 \text{ or } 1 \end{bmatrix}$$

Shear: Intermixes coordinates according to $\alpha, \beta \in \mathbb{R}$:

- Applies to points or vectors, is linear.
- Matrix representation:

$$\overbrace{\begin{bmatrix} 1 & \beta & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}^{Sh(\alpha,\beta)} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x + \beta y \\ \alpha x + y \\ 0 \text{ or } 1 \end{bmatrix}$$

- Easiest to see if we set one of α or β to zero.

Reflection: Through a line.

- Applies to points or vectors, is linear.
- Example: through x-axis, matrix representation is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \text{ or } 1 \end{bmatrix} = \begin{bmatrix} x \\ -y \\ 0 \text{ or } 1 \end{bmatrix}$$

- See book for other examples

Note: Vectors map through all these transformations as we want them to.

(Readings: Hearn and Baker, 5-1, 5-2, and 5-4; Red book, 5.2, 5.3; White book, 5.1, 5.2)

6.6 Compositions of Transformations

Compositions of Transformations

Suppose we want to rotate around an arbitrary point $P \equiv [x, y, 1]^T$.

- Could derive a more general transformation matrix ...
- Alternative idea: Compose simple transformations
  1. Translate P to origin
  2. Rotate around origin
  3. Translate origin back to P
- Suppose $P = [x_o, y_o, 1]^T$
- The the desired transformation is

$$\begin{aligned} & T(x_o, y_o) \circ R(\theta) \circ T(-x_o, -y_o) \\ = & \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \circ \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \circ \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \\ = & \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x_o(1-\cos(\theta)) + y_o \sin(\theta) \\ \sin(\theta) & \cos(\theta) & y_o(1-\cos(\theta)) - x_o \sin(\theta) \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

- Note where P maps: to P.
- Won't compute these matrices analytically;
  Just use the basic transformations,
  and run the matrix multiply numerically.

Order is important!

$$\begin{aligned} & T(-\Delta x, -\Delta y) \circ T(\Delta x, \Delta y) \circ R(\theta) = R(\theta) \\ & \neq T(-\Delta x, -\Delta y) \circ R(\theta) \circ T(\Delta x, \Delta y). \end{aligned}$$

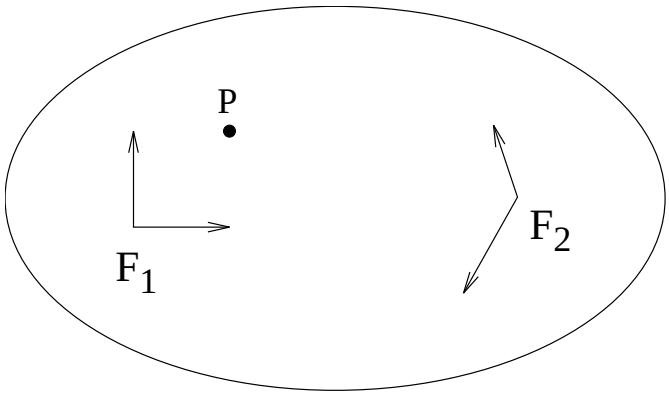
(Readings: Hearn and Baker, Section 5-3; Red book, 5.4; White book, 5.3)

6.7 Change of Basis

Change of Basis

Suppose:

- We have two coordinate frames for a space, F_1 and F_2,
- Want to change from coordinates relative to F_1 to coordinates relative to F_2.



Know $P \equiv \mathbf{p} = [x, y, 1]^T$ relative to $F_1 = (\vec{w}_1, \vec{w}_2, \mathcal{O}_W)$.

Want the coordinates of P relative to $F_2 = (\vec{v}_1, \vec{v}_2, \mathcal{O}_V)$.

How do we get $f_{i,j}$?

- If F_2 is orthonormal:

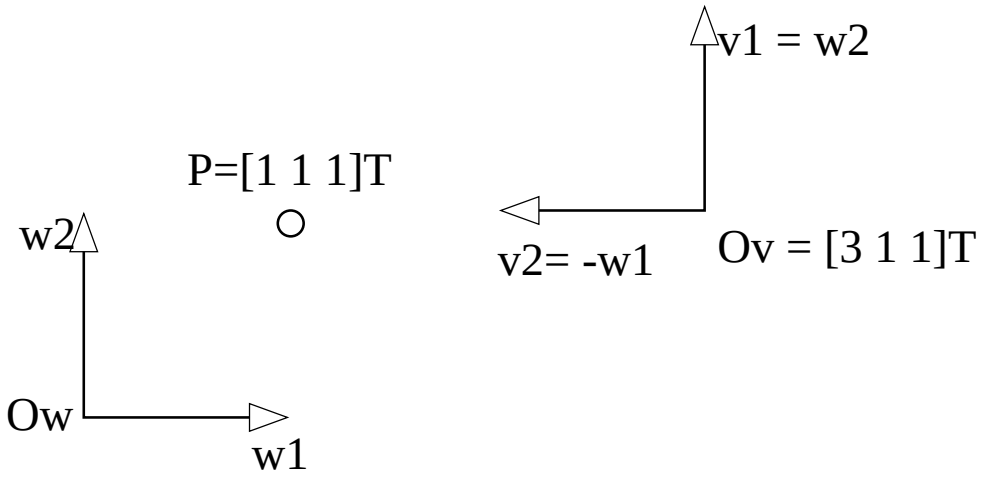
$$f_{i,j} = \vec{w}_j \cdot \vec{v}_i,$$
$$f_{i,3} = (\mathcal{O}_W - \mathcal{O}_V) \cdot \vec{v}_i.$$

- If F_2 is orthogonal:

$$f_{i,j} = \frac{\vec{w}_j \cdot \vec{v}_i}{\vec{v}_i \cdot \vec{v}_i},$$
$$f_{i,3} = \frac{(\mathcal{O}_W - \mathcal{O}_V) \cdot \vec{v}_i}{\vec{v}_i \cdot \vec{v}_i}.$$

- Otherwise, we have to solve a small system of linear equations, using $F_1 = F_2 M_{1,2}$.
- Change of basis from F_1 to Standard Cartesian Frame is trivial (since frame elements normally expressed with respect to Standard Cartesian Frame).

Example:



where F_W is the standard coordinate frame.

Generalization to 3D is straightforward ...

Example:

- Define two frames:

$$\begin{aligned} F_W &= (\vec{w}_1, \vec{w}_2, \vec{w}_3, \mathcal{O}_W) \\ &= \left(\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right) \\ F_V &= (\vec{v}_1, \vec{v}_2, \vec{v}_3, \mathcal{O}_V) \\ &= \left(\begin{bmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} \sqrt{2}/2 \\ -\sqrt{2}/2 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 3 \\ 1 \end{bmatrix} \right) \end{aligned}$$

  - All coordinates are specified relative to the standard frame in a Cartesian 3 space.

- In this example, F_W is the standard frame.
- Note that both F_W and F_V are orthonormal.
- The matrix mapping F_W to F_V is given by

$$M = \begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 & 0 & -\sqrt{2}/2 \\ 0 & 0 & 1 & -3 \\ \sqrt{2}/2 & -\sqrt{2}/2 & 0 & -\sqrt{2}/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Question: What is the matrix mapping from F_V to F_W?

Notes

- On the computer, frame elements usually specified in Standard Frame for space.
  Eg, a frame $F = [\vec{v}_1,\ \vec{v}_2,\ \mathcal{O}_V]$ is given by

$$[\ [v_{1x},\ v_{1y},\ 0]^T,\ \ [v_{2x},\ v_{2y},\ 0]^T,\ \ [v_{3x},\ v_{3y},\ 1]^T\ ]$$

  relative to Standard Frame.
  Question: What are coordinates of these basis elements relative to F?
- Frames are usually orthonormal.
- A point "mapped" by a change of basis does *not change*;
  We have merely expressed its coordinates relative to a different frame.

(Readings: Watt: 1.1.1. Hearn and Baker: Section 5-5 (not as general as here, though). Red book: 5.9. White book: 5.8.)

6.8 Ambiguity

Ambiguity

Three Types of Transformations:

1. $T : \mathcal{A} \mapsto \mathcal{B}$ (between two spaces)
2. $T : \mathcal{A} \mapsto \mathcal{A}$ ("warp" an object within its own space)
3. T : change of coordinates

Changes of Coordinates:

- Given 2 frames:
  - $F_1 = (\vec{v}_1, \vec{v}_2, \mathcal{O})$, orthonormal,
  - $F_2 = (2\vec{v}_1, 2\vec{v}_2, \mathcal{O})$, orthogonal.
- Suppose $\vec{v} \equiv F_1[1, 1, 0]^T$.
- Then $\vec{v} \equiv F_2[1/2, 1/2, 0]^T$.

Question: What is the length of $\vec{v}$?

Suppose we have $P \equiv [p_1, p_2, 1]^T$ and $Q \equiv [q_1, q_2, 1]^T$

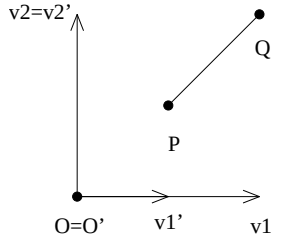
- P, Q relative to $F = (\vec{v}_1, \vec{v}_2, \mathcal{O})$,
- We are given a matrix representation of a transformation T:

$$M_T = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

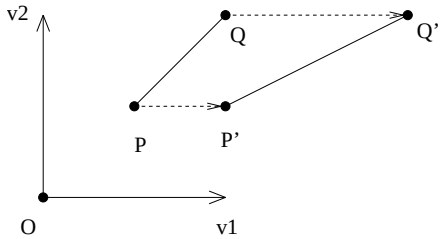
- Consider $P' = TP$ and $Q' = TQ$.
- How do we interpret P' and Q'?

How do we interpret P' and Q'?

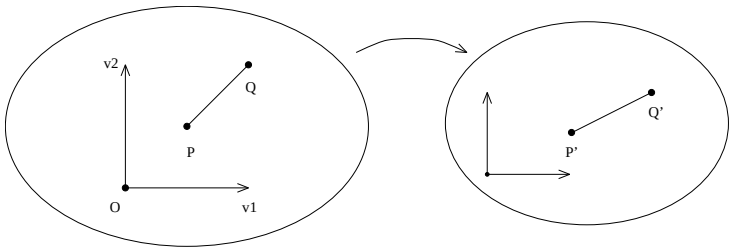
1. Change of Coordinates?



2. Scale?



3. Transformations between spaces?



Do we care? YES!

- In (1) nothing changes except the representation.
- In (1) distances are preserved while they change in (2) and the question has no meaning in (3).
- In (3), we've completely changed spaces.

Consider the meaning of $|P' - P|$

1. $|P' - P| = 0$
2. $|P' - P| = \sqrt{(2p_1 - p_1)^2 + (p_2 - p_2)^2} = |p_1|$
3. $|P' - P|$ has no meaning

To fully specify a transformation, we need

1. A matrix
2. A domain space
3. A range space
4. A coordinate frame in each space

6.9 3D Transformations

3D Transformations

Assume a right handed coordinate system

Points $P \equiv [x, y, z, 1]^T$, Vectors $\vec{v} \equiv [x, y, z, 0]^T$

Translation:

$$T(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale: About the origin

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation: About a coordinate axis

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

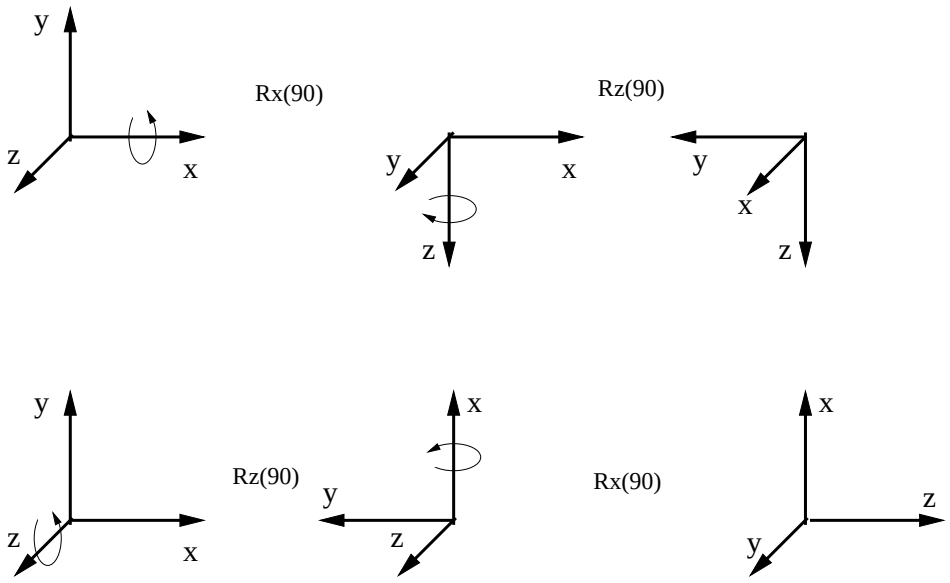
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear: see book

Reflection: see book

Composition: works same way (but order counts when composing rotations).



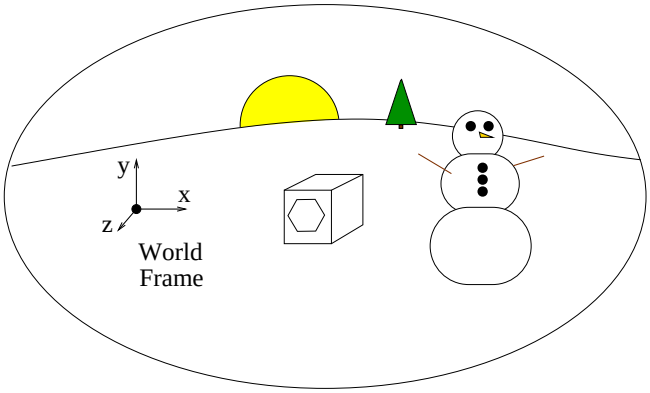
(Readings: Hearn and Baker, Chapter 11; Red book, 5.7; White book, 5.6)

6.10 World and Viewing Frames

World and Viewing Frames

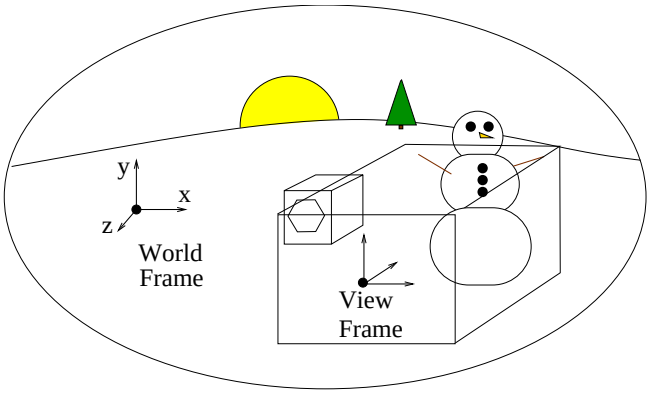
- Typically, our space S is a Cartesian space.
  - Call the standard frame the **world frame**.
  - The world frame is typically **right handed**.

  - Our scene description is specified in terms of the world frame.



- The viewer may be anywhere and looking in any direction.
  - Often, x to the right, y up, and z straight ahead.
    - z is called the *viewing direction*.
    - This is a **left handed** coordinate system.
  - We could instead specify z and y as vectors
    - z is the the **view direction**.
    - y is the *up vector*.
    - Compute $x = y \times z$
    - Get a **right handed** coordinate system.
  - We can do a change of basis
    - Specify a frame relative to the viewer.
    - Change coordinates to this frame.

- Once in viewing coordinates,
  - Usually place a clipping "box" around the scene.
  - Box oriented relative to the viewing frame.



- An **orthographic projection** is made by "removing the $z-$coordinate."

- Squashes 3D onto 2D, where we can do the window-to-viewport map.
- The projection of the clipping box is used as the window.

- Mathematically, relative to
$$F_V = (\vec{\imath}, \vec{\jmath}, \vec{k}, \mathcal{O})$$
we map $Q \equiv [q_1, q_2, q_3, 1]^T$ onto
$$F_P = (\vec{u}, \vec{v}, \mathcal{O}')$$
as follows:
$$\text{Ortho}(q_1\vec{\imath} + q_2\vec{\jmath} + q_3\vec{k} + \mathcal{O}) = q_1\vec{u} + q_2\vec{v} + \mathcal{O}'$$
or if we ignore the frames,
$$[q_1, q_2, q_3, 1]^T \mapsto [q_1, q_2, 1]^T$$

- We can write this in matrix form:
$$\begin{bmatrix} q_1 \\ q_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ 1 \end{bmatrix}$$

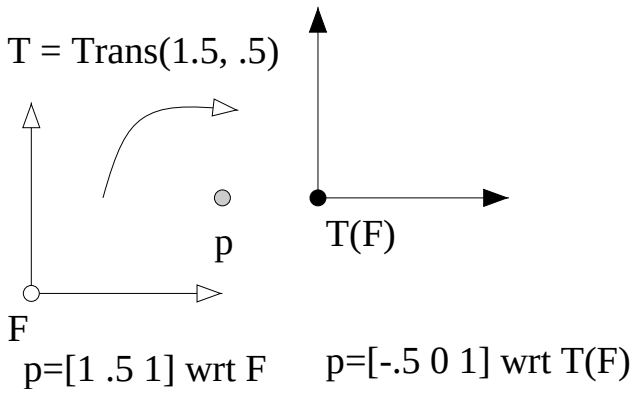
- Question: why would we want to write this in matrix form?

Viewing-World-Modelling Transformations :

- Want to do modelling transformations and viewing transformation (as in Assignment 2).
- If V represents World-View transformation, and M represents modelling transformation, then
$$VM$$
transforms from modelling coordinates to viewing coordinates.
Note: M is performing both modelling transformation and Model to World change of basis.
- Question: If we transform the viewing frame (relative to viewing frame) how do we adjust V?
- Question: If we transform model (relative to modelling frame) how do we adjust M?

Viewing Transformations:

- Assume all frames are orthonormal
- When we transform the View Frame by T, apply T^{-1} to anything expressed in old view frame coordinates to get new View Frame coordinates



- To compute new World-to-View change of basis, need to express World Frame in new View Frame

  Get this by transforming World Frame elements represented in old View Frame by T^{-1}.

- Recall that the columns of the World-to-View change-of-basis matrix are the basis elements of the World Frame expressed relative to the View Frame.
- If V is old World-to-View change-of-basis matrix, then $T^{-1}V$ will be new World-to-View change-of-basis matrix, since each column of V represents World Frame element, and the corresponding column of $T^{-1}V$ contains T^{-1} of this element.

Modelling Transformations:

- Note that the columns of M are the Model Frame elements expressed relative to the World Frame.
- Want to perform modelling transformation relative to modelling coordinates.
- If we have previously transformed Model Frame, then we next transform relative to transformed Model Frame.
- Example: If

$$M = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  and we translate one unit relative to the first Model Frame basis vector, then we want to translate by $(x_1, y_1, z_1, 0)$ relative to the World Frame.

- Could write this as

$$M' = \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot M = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 + x_1 \\ y_1 & y_2 & y_3 & y_4 + y_1 \\ z_1 & z_2 & z_3 & z_4 + z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- But this is also equal to

$$M' = M \cdot \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 + x_1 \\ y_1 & y_2 & y_3 & y_4 + y_1 \\ z_1 & z_2 & z_3 & z_4 + z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- In general, if we want to transform by T our model relative to the current Model Frame, then

$$MT$$

  yields that transformation.
- Summary:

  Modelling transformations embodied in matrix M

  World-to-View change of basis in matrix V

  VM transforms from modelling coordinates to viewing coordinates

  If we further transform the View Frame by T relative to the View Frame, then the new change-of-basis matrix V' is given by

$$V' = T^{-1}V$$

  If we further transform the model by T relative to the modelling frame, the new modelling transformation M' is given by

$$M' = MT$$

- For Assignment 2, need to do further dissection of transformations, but this is the basic idea.

(Readings: Hearn and Baker, Section 6-2, first part of Section 12-3; Red book, 6.7; White book, 6.6)

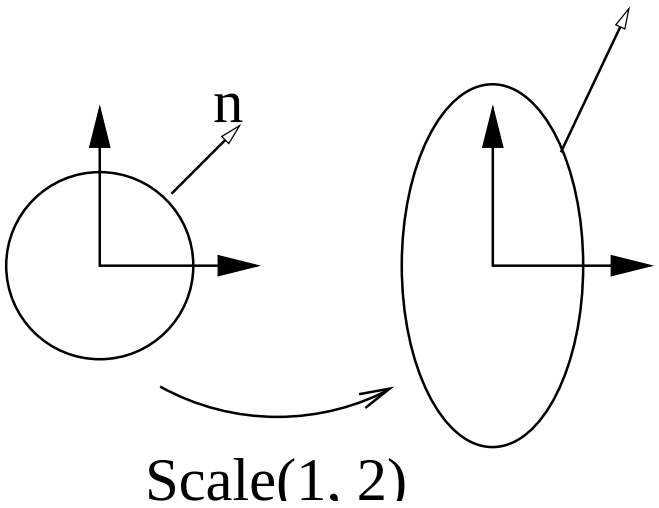
6.11 Normals

Transforming Normals

The Truth: Can really only apply affine transforms to points.
Vectors can be transformed correctly iff they are defined by differences of points.

Transforming Normal Vectors:

Consider non-uniform scale of circle, and normal to circle:



Why doesn't normal transform correctly?

- Normal vectors **ARE NOT** defined by differences of points (formally, they are *covectors*, which are *dual* to vectors).
- Tangent vectors **ARE** defined by differences of points.
- Normals are vectors perpendicular to all tangents at a point:

$$\vec{N} \cdot \vec{T} \equiv \mathbf{n}^T \mathbf{t} = 0.$$

- Note that the natural representation of $\vec{N}$ is as a *row vector*.
- Suppose we have a transformation M, a point $P \equiv \mathbf{p}$, and a tangent $\vec{T} \equiv \mathbf{t}$ at P.
- Let M_ℓ be the "linear part" of M, i.e. the upper 3×3 submatrix.

$$\begin{aligned}
\mathbf{p}' &= M\mathbf{p}, \\
\mathbf{t}' &= M\mathbf{t} \\
&= M_\ell \mathbf{t}. \\
\mathbf{n}^T\mathbf{t} &= \mathbf{n}^T M_\ell^{-1} M_\ell \mathbf{t} \\
&= (M_\ell^{-1T}\mathbf{n})^T (M_\ell \mathbf{t}) \\
&= (\mathbf{n}')^T \mathbf{t}' \\
&\equiv \vec{N}' \cdot \vec{T}'.
\end{aligned}$$

- Transform normals by inverse transpose of linear part of transformation: $\mathbf{n}' = M_\ell^{-1T}\mathbf{n}$.
- If M_T is O.N. (usual case for rigid body transforms), $M_T^{-1T} = M_T$.
- Only worry if you have a non-uniform scale or a shear transformation.
- Transforming lines: Transform implicit form in a similar way.
- Transforming planes: Transform implicit form in a similar way.

(Readings: Red Book, 5.8 (?); White Book, 5.6.)

7 Windows, Viewports, NDC

7.1 Window to Viewport Mapping

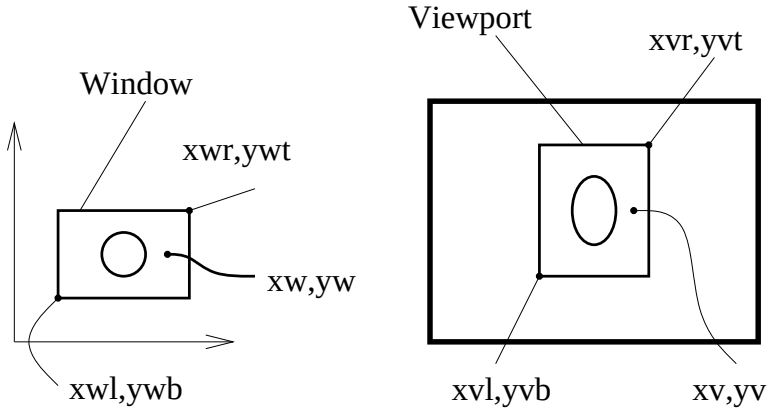
Window to Viewport Mapping

- Start with 3D scene, but eventually project to 2D scene
- 2D scene is infinite plane. Device has a finite visible rectangle.
  What do we do?
- Answer: map rectangular region of 2D device scene to device.

  Window: rectangular region of interest in scene.

  Viewport: rectangular region on device.

  Usually, both rectangles are aligned with the coordinate axes.



- Window point (x_w, y_w) maps to viewport point (x_v, y_v).

  Length and height of the window are L_w and H_w,
  Length and height of the viewport are L_v and H_v.
- Proportionally map each of the coordinates according to:

$$\frac{\Delta x_w}{L_w} = \frac{\Delta x_v}{L_v}, \qquad \frac{\Delta y_w}{H_w} = \frac{\Delta y_v}{H_v}.$$

- To map x_w to x_v:

$$\begin{aligned} \frac{x_w - x_{wl}}{L_w} &= \frac{x_v - x_{vl}}{L_v} \\ \Rightarrow \quad x_v &= \frac{L_v}{L_w}(x_w - x_{wl}) + x_{vl}, \end{aligned}$$

  and similarily for y_v.
- If $H_w/L_w \neq H_v/L_v$ the image will be distorted.
  These quantities are called the **aspect ratios** of the window and viewport.

(Readings: Watt: none. Hearn and Baker: Section 6-3. Red book: 5.5. White book: 5.4, Blinn: 16.

Intuitively, the window-to-viewport formula can be read as:

- Convert x_w to a distance from the window corner.
- Scale this w distance to get a v distance.
- Add to viewport corner to get x_v.

)

7.2 Normalized Device Coordinates

Normalized Device Coordinates

- Where do we specify our viewport?
- Could specify it in device coordinates ...
  BUT, suppose we want to run on several platforms/devices

  Two common conventions for DCS:

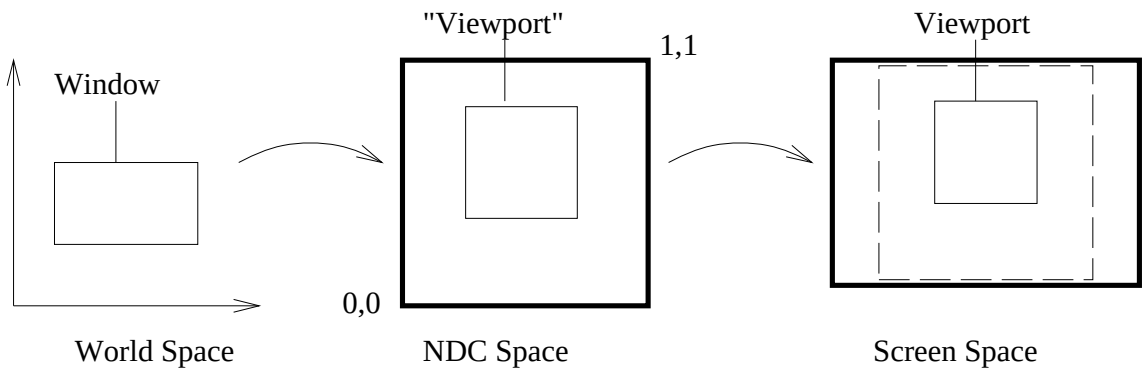
  - Origin in the lower left, with x to the right and y upward.
  - Origin in the top left, with x to the right and y downward.

  Many different resolutions for graphics display devices:

  - Workstations commonly have 1280×1024 frame buffers.
  - A PostScript page is 612×792 points, but 2550×3300 pixels at 300dpi.
  - And so on ...

  Aspect ratios may vary ...

- If we map directly from WCS to a DCS, then changing our device requires rewriting this mapping (among other changes).
- Instead, use **Normalized Device Coordinates (NDC)** as an intermediate coordinate system that gets mapped to the device layer.
- Will consider using only a square portion of the device.
  Windows in WCS will be mapped to viewports that are specified within a unit square in NDC space.
- Map viewports from NDC coordinates to the screen.



(Readings: Watt: none. Hearn and Baker: Sections 2-7, 6-3. Red book: 6.3. White book: 6.5.)

8 Clipping

8.1 Clipping

Clipping

Clipping: Remove points outside a region of interest.

- Discard (parts of) primitives outside our window...

Point clipping: Remove points outside window.

- A point is either entirely inside the region or not.

Line clipping: Remove portion of line segment outside window.

- Line segments can straddle the region boundary.
- Liang-Barsky algorithm efficiently clips line segments to a halfspace.
- Halfspaces can be combined to bound a convex region.
- Can use some of the ideas in Liang-Barsky to clip points.

Parametric representation of line:

$$L(t) = (1-t)A + tB$$

or equivalently

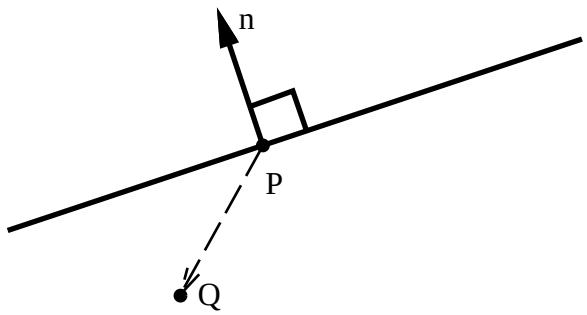
$$L(t) = A + t(B-A)$$

- A and B are non-coincident points.
- For $t \in \mathbb{R}$, $L(t)$ defines an infinite line.
- For $t \in [0,1]$, $L(t)$ defines a line segment from A to B.
- Good for generating points on a line.
- Not so good for testing if a given point is on a line.

Implicit representation of line:

$$\ell(Q) = (Q-P) \cdot \vec{n}$$

- P is a point on the line.
- $\vec{n}$ is a vector perpendicular to the line.
- $\ell(Q)$ gives us the signed distance from any point Q to the line.
- The sign of $\ell(Q)$ tells us if Q is on the left or right of the line, relative to the direction of $\vec{n}$.
- If $\ell(Q)$ is zero, then Q is on the line.
- Use same form for the implicit representation of a halfspace.



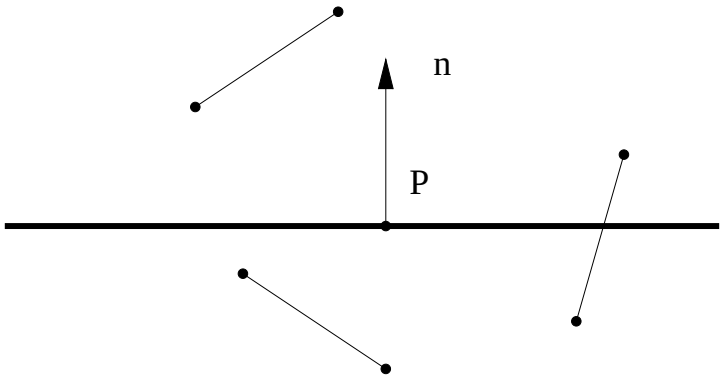
Clipping a point to a halfspace:

- Represent window edge as implicit line/halfspace.
- Use the implicit form of edge to classify a point Q.
- Must choose a convention for the normal:
  points to the *inside.*
- Check the sign of $\ell(Q)$:
  - If $\ell(Q) > 0$, then Q is inside.
  - Otherwise clip (discard) Q:
    It is on the edge or outside.
    May want to keep things on the boundary.

Clipping a line segment to a halfspace:

There are three cases:

1. The line segment is entirely inside:
   Keep it.
2. The line segment is entirely outside:
   Discard it.
3. The line segment is partially inside and partially outside:
   Generate new line to represent part inside.



Input Specification:

- Window edge: implicit, $\ell(Q) = (Q - P) \cdot \vec{n}$
- Line segment: parametric, $L(t) = A + t(B - A)$.

Do the easy stuff first:

We can devise easy (and fast!) tests for the first two cases:

- $\ell(A) < 0$ **AND** $\ell(B) < 0 \Longrightarrow$ Outside
- $\ell(A) > 0$ **AND** $\ell(B) > 0 \Longrightarrow$ Inside

Need to decide: are **boundary points inside** or **outside**?

Trivial tests are important in computer graphics:

- Particularly if the trivial case is the most common one.
- Particularly if we can reuse the computation for the non-trivial case.

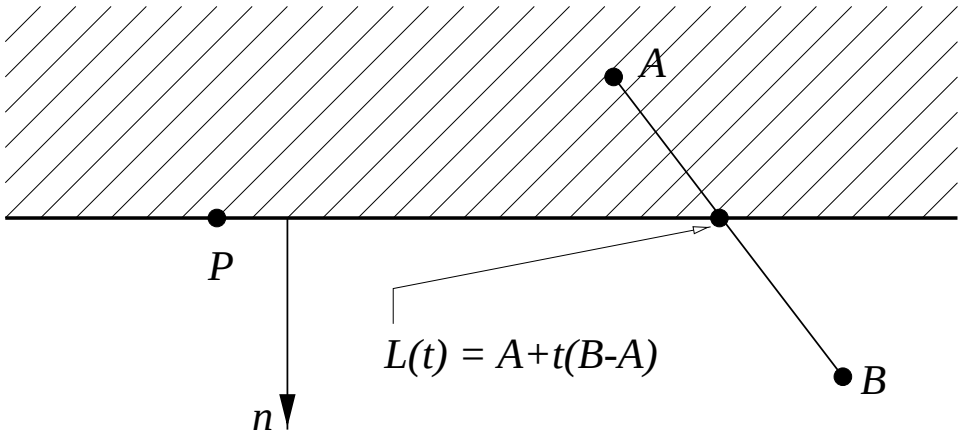
Do the hard stuff only if we have to:

If line segment partially in/partially out, need to clip it:

- Line segment from A to B in **parametric** form:

$$L(t) = (1 - t)A + tB = A + t(B - A)$$

- When $t = 0$, $L(t) = A$. When $t = 1$, $L(t) = B$.
- We now have the following:



Recall: $\ell(Q) = (Q - P) \cdot \vec{n}$

- We want t such that $\ell(L(t)) = 0$:

$$\begin{aligned}(L(t) - P) \cdot \vec{n} &= (A + t(B - A) - P) \cdot \vec{n} \\ &= (A - P) \cdot \vec{n} + t(B - A) \cdot \vec{n} \\ &= 0\end{aligned}$$

- Solving for t gives us

$$t = \frac{(A - P) \cdot \vec{n}}{(A - B) \cdot \vec{n}}$$

- **NOTE:**
  The values we use for our simple test can be reused to compute t:

$$t = \frac{(A-P)\cdot\vec{n}}{(A-P)\cdot\vec{n}-(B-P)\cdot\vec{n}}$$

Clipping a line segment to a window:
Just clip to each of four halfspaces in turn.

Pseudo-code (here wec = *window-edge coordinates*):

```
Given line segment (A,B), clip in-place:
for each edge (P,n)
    wecA = (A-P) . n
    wecB = (B-P) . n
    if ( wecA < 0 AND wecB < 0 ) then reject
    if ( wecA >= 0 AND wecB >= 0 ) then next
    t = wecA / (wecA - wecB)
    if ( wecA < 0 ) then
        A = A + t*(B-A)
    else
        B = A + t*(B-A)
    endif
endfor
```

Note:

- Liang-Barsky Algorithm can clip lines to any **convex** window.
- Optimizations can be made for the special case of horizontal and vertical window edges.

Question:
Should we clip before or after window-to-viewport mapping?

Line-clip Algorithm generalizes to 3D:

- Half-space now lies on one side of a *plane*.
- Plane also given by normal and point.
- Implicit formula for plane in 3D is same as that for line in 2D.
- Parametric formula for line to be clipped is unchanged.

(Readings: Watt: none. Hearn and Baker: Sections 6-5 through 6-7 (12-5 for 3D clipping). Red book: 3.9. White Book: 3.11. Blinn: 13.)

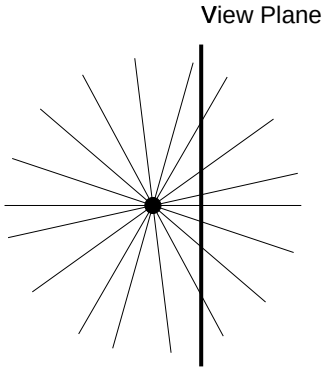
9 Projections and Projective Transformations

9.1 Projections

Projections

Perspective Projection

- Identify all points with a line through the eyepoint.
- Slice lines with viewing plane, take intersection point as projection.



- This is **not** an affine transformation, but a **projective transformation**.

Projective Transformations:

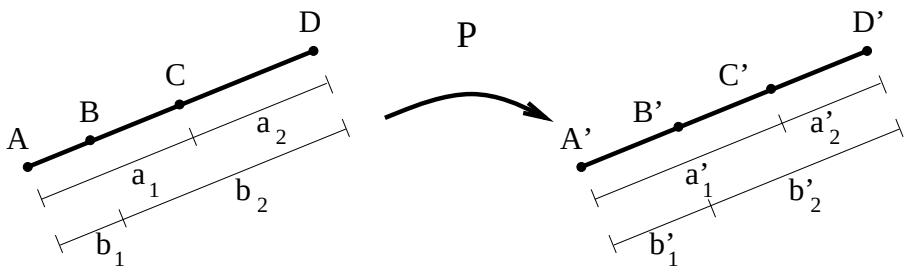
- Angles are not preserved (not preserved under Affine Transformation).
- Distances are not preserved (not preserved under Affine Transformation).
- Ratios of distances are not preserved.
- Affine combinations are not preserved.
- Straight lines are mapped to straight lines.
- *Cross ratios* are preserved.

Cross Ratios

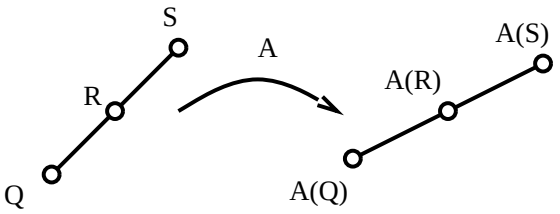
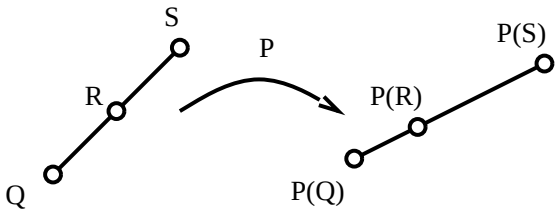
- Cross ratio: $|AC| = a_1$, $|CD| = a_2$, $|AB| = b_1$, $|BD| = b_2$, then

$$\frac{a_1/a_2}{b_1/b_2} \left(= \frac{a_1'/a_2'}{b_1'/b_2'} \right)$$

This can also be used to define a projective transformation (ie, that lines map to lines and cross ratios are preserved).

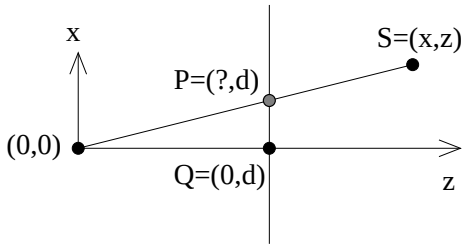


Comparison:

Affine Transformations	**Projective Transformations**
Image of 2 points on a line determine image of line	Image of 3 points on a line determine image of line
Image of 3 points on a plane determine image of plane	Image of 4 points on a plane determine image of plane
In dimension n space, image of $n+1$ points/vectors defines affine map.	In dimension n space, image of $n+2$ points/vectors defines projective map.
Vectors map to vectors $\vec{v} = Q - R = R - S \Rightarrow$ $A(Q) - A(R) = A(R) - A(S)$	Mapping of vector is ill-defined $\vec{v} = Q - R = R - S$ but $P(Q) - P(R) \neq P(R) - P(S)$
	
Can represent with matrix multiply (sort of)	Can represent with matrix multiply and normalization

Perspective Map:

- Given a point S, we want to find its projection P.

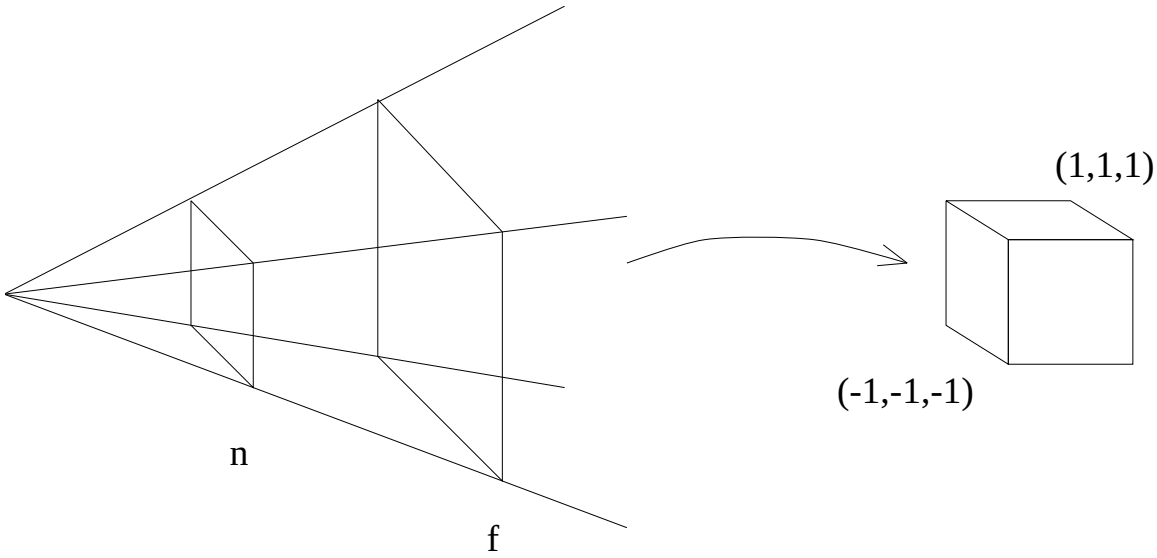


- Similar triangles: $P = (xd/z, d)$
- In 3D, $(x, y, z) \mapsto (xd/z, yd/z, d)$
- Have identified all points on a line through the origin with a point in the projection plane.
  $(x, y, z) \equiv (kx, ky, kz),\ k \neq 0.$
- These are known as homogeneous coordinates.
- If we have solids or coloured lines,
  then we need to know "which one is in front".
- This map loses all z information, so it is inadequate.

Pseudo-OpenGL version of the perspective map:

- Maps a near clipping plane $z = n$ to $z' = -1$

- Maps a far clipping plane $z = f$ to $z' = 1$



- The "box" in world space known as "truncated viewing pyramid" or "frustum"
  - Project x, y as before
  - To simplify things, we will project into the $z = 1$ plane.

Derivation:

- Want to map x to x/z (and similarly for y).
- Use matrix multiply followed by **homogenization**:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & c \\ 0 & 0 & b & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ az + c \\ bz + d \end{bmatrix}$$

$$\equiv \begin{bmatrix} \frac{x}{bz+d} \\ \frac{y}{bz+d} \\ \frac{az+c}{bz+d} \\ 1 \end{bmatrix}$$

- Solve for a, b, c, and d such that $z \in [n, f]$ maps to $z' \in [-1, 1]$.

- Satisfying our constraints gives us

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{z(f+n)-2fn}{f-n} \\ z \end{bmatrix}$$

- After homogenizing we get

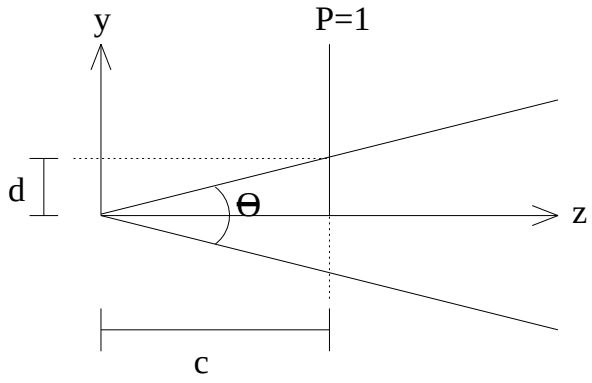
$$[\frac{x}{z}, \frac{y}{z}, \frac{z(f+n) - 2fn}{z(f-n)}, 1]^T$$

- Could use this formula instead of performing the matrix multiply followed by the division ...
- If we multiply this matrix in with the geometric transforms,
  the only additional work is the divide.

The OpenGL perspective matrix uses

- $a = -\frac{f+n}{f-n}$ and $b = -1$.
  - OpenGL looks down $z = -1$ rather than $z = 1$.
  - Note that when you specify n and f,
    they are given as *positive* distances down $z = -1$.
- The upper left entries are very different.
  - OpenGL uses this one matrix to both project and map to NDC.
  - How do we set x or y to map to $[-1, 1]$?
  - We don't want to do both because we may not have square windows.

OpenGL maps y to $[-1, 1]$:



- Want to map distance d to 1.
- $y \mapsto y/z$ is the current projection ...

Our final matrix is

$$\begin{bmatrix} \frac{\cot(\theta/2)}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\theta/2) & 0 & 0 \\ 0 & 0 & \pm\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & \pm 1 & 0 \end{bmatrix}$$

where the $\pm$ is 1 if we look down the z axis and -1 if we look down the $-z$ axis.

OpenGL uses a slightly more general form of this matrix that allows skewed viewing pyramids.

9.2 Why Map Z?

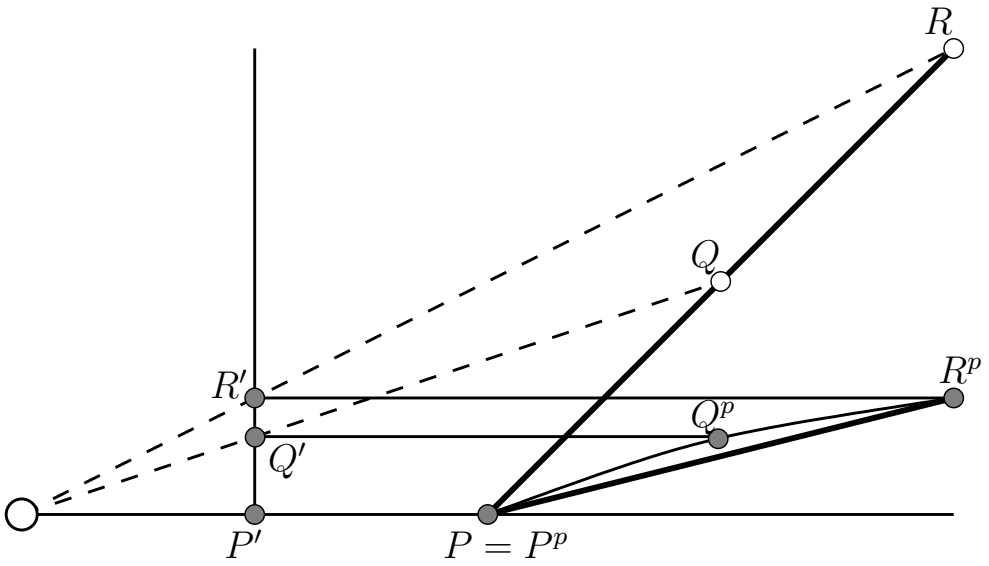
Why Map Z?

- 3D $\mapsto$ 2D projections map all z to same value.
- Need z to determine occlusion, so a 3D to 2D projective transformation doesn't work.
- Further, we want 3D lines to map to 3D lines (this is useful in hidden surface removal)
- The mapping $(x, y, z, 1) \mapsto (xn/z, yn/z, n, 1)$ maps lines to lines, but loses all depth information.
- We could use

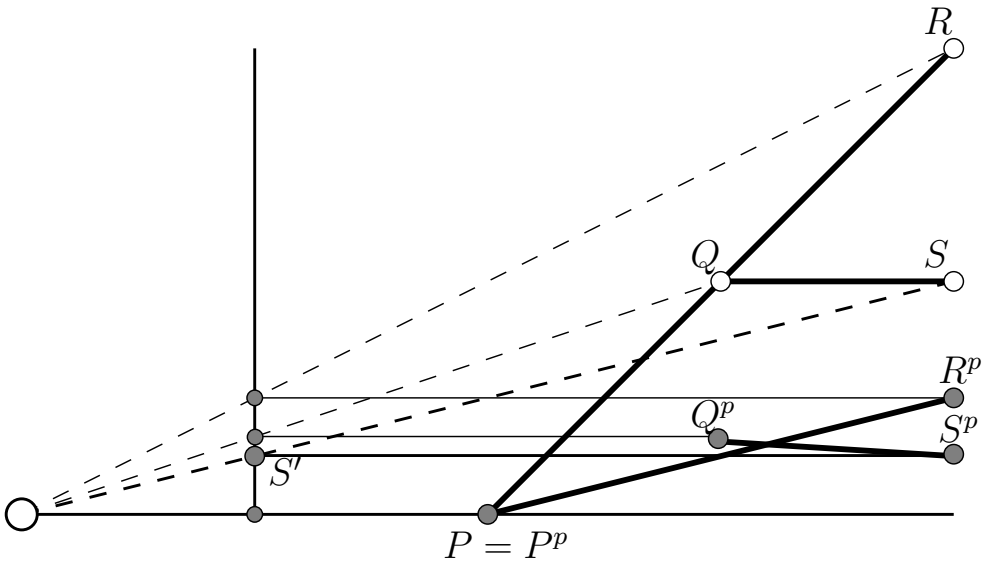
$$(x, y, z, 1) \quad \mapsto \quad (\frac{xn}{z}, \frac{yn}{z}, z, 1)$$

Thus, if we map the endpoints of a line segment, these end points will have the same relative depths after this mapping.

BUT: It fails to map lines to lines



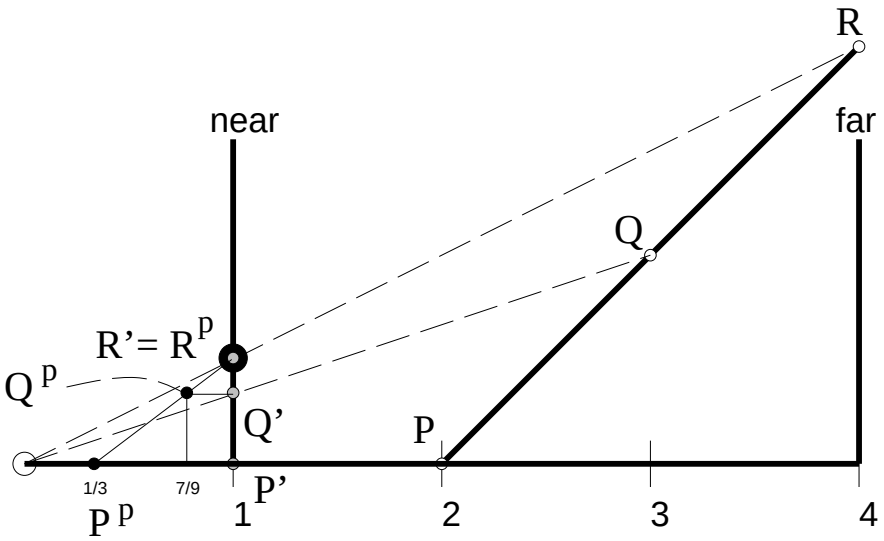
- In this figure, P, Q, R map to P', Q', R' under a pure projective transformation.
- With the mapping $(x, y, z, 1) \mapsto (\frac{xn}{z}, \frac{yn}{z}, z, 1)$ P, Q, R actually map to P^p, Q^p, R^p, which fail to lie on a straight line.



- Now map S to S' to S^p.
- Line segments P^pR^p and Q^pS^p cross!
- The map

$$(x, y, z, 1) \quad \mapsto \quad (\frac{xn}{z}, \frac{yn}{z}, \frac{zf + zn - 2fn}{z(f - n)}, 1)$$

  does map lines to lines, **and** it preserves depth information.



9.3 Mapping Z

Mapping Z

- It's clear how x and y map. How about z?

- The z map affects: clipping, numerics

$$z \quad \mapsto \quad \frac{zf + zn - 2fn}{z(f-n)} = P(z)$$

- We know $P(f) = 1$ and $P(n) = -1$. What maps to 0?

$$\begin{array}{rcl} & P(z) & = 0 \\ \Rightarrow & \frac{zf+zn-2fn}{z(f-n)} & = 0 \\ \Rightarrow & z & = \dfrac{2fn}{f+n} \end{array}$$

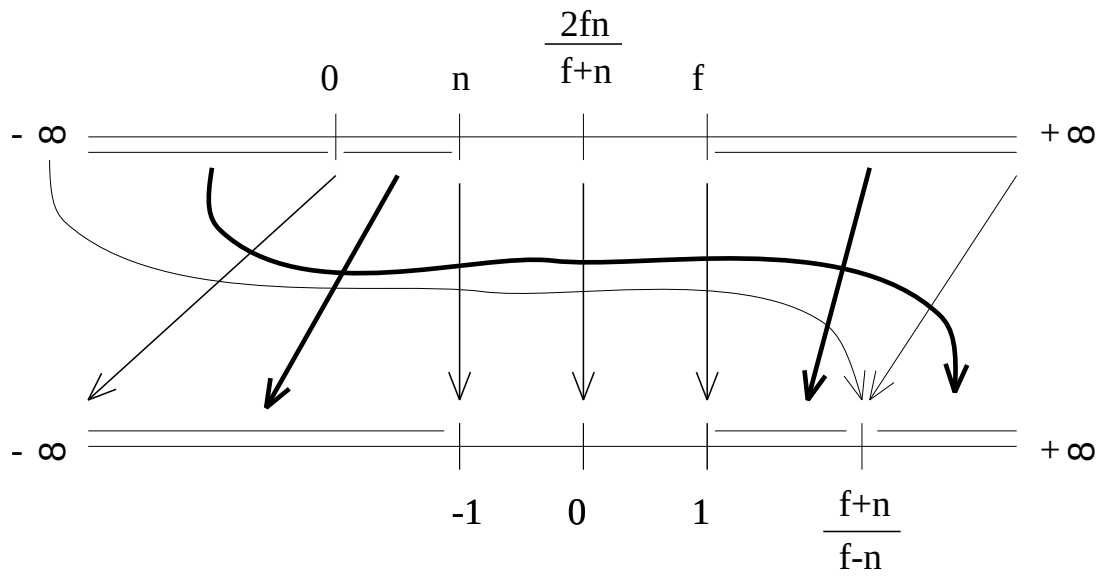
  Note that $f^2 + fn > 2fn > fn + n^2$ so

$$f > \frac{2fn}{f+n} > n$$

- What happens as z goes to 0 or to infinity?

$$\begin{array}{rcl} \lim\limits_{z\to 0^+} P(z) & = & \dfrac{-2fn}{z(f-n)} \\ & = & -\infty \\ \lim\limits_{z\to 0^-} P(z) & = & \dfrac{-2fn}{z(f-n)} \\ & = & +\infty \\ \lim\limits_{z\to +\infty} P(z) & = & \dfrac{z(f+n)}{z(f-n)} \\ & = & \dfrac{f+n}{f-n} \\ \lim\limits_{z\to -\infty} P(z) & = & \dfrac{z(f+n)}{z(f-n)} \\ & = & \dfrac{f+n}{f-n} \end{array}$$

Pictorially, we have



- What happens if we vary f and n?

  –

  $$\begin{aligned}\lim_{f\to n} P(z) &= \frac{z(f+n)-2fn}{z(f-n)} \\ &= \frac{(2zn-2n^2)}{z\cdot 0}\end{aligned}$$

  –

  $$\begin{aligned}\lim_{f\to \infty} P(z) &= \frac{zf-2fn}{zf} \\ &= \frac{z-2n}{z}\end{aligned}$$

  –

  $$\begin{aligned}\lim_{n->0} P(z) &= \frac{zf}{zf} \\ &= 1\end{aligned}$$

- What happens as f and n move away from each other.
  Look at size of the regions $[n, 2fn/(f+n)]$ and $[2fn/(f+n), f]$.

- When f is large compared to n, we have

  $$\frac{2fn}{f+n} \doteq 2n$$

  So

  $$\frac{2fn}{f+n} - n \doteq n$$

and

$$f - \frac{2fn}{f+n} \doteq f - 2n.$$

But both intervals are mapped to a region of size 1.

9.4 3D Clipping

3D Clipping

- When do we clip in 3D?

  We should clip to the near plane *before* we project. Otherwise, we might attempt to project a point with $z = 0$ and then x/z and y/z are undefined.

- We could clip to all 6 sides of the truncated viewing pyramid.

  But the plane equations are simpler if we clip after projection, because all sides of volume are parallel to coordinate plane.

- Clipping to a plane in 3D is identical to clipping to a line in 2D.

- We can also clip in homogeneous coordinates.

(Readings: Red Book, 6.6.4; White book, 6.5.4.)

9.5 Homogeneous Clipping

Homogeneous Clipping

Projection: transform and homogenize

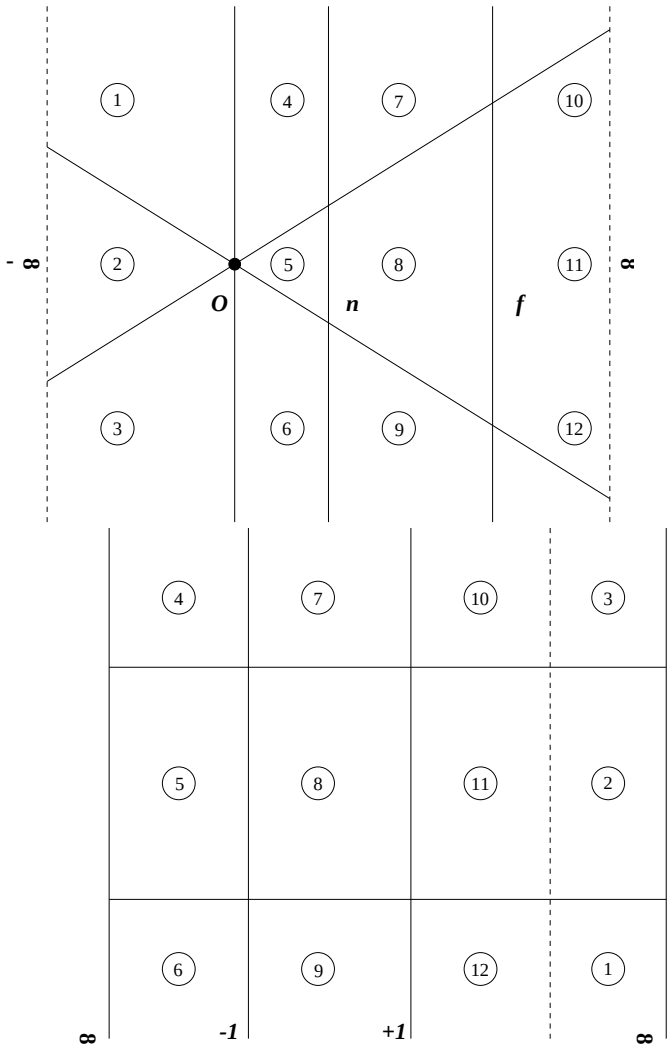
- Linear transformation

$$\begin{bmatrix} nr & 0 & 0 & 0 \\ 0 & ns & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix}$$

- Homogenization

$$\begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \\ \bar{w} \end{bmatrix} = \begin{bmatrix} \bar{x}/\bar{w} \\ \bar{y}/\bar{w} \\ \bar{z}/\bar{w} \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Region mapping:



Clipping not good after homogenization:

- Ambiguity after homogenization

$$-1 \leq \frac{\bar{x}, \bar{y}, \bar{z}}{\bar{w}} \leq +1$$

  - Numerator can be positive or negative
  - Denominator can be positive or negative
- Normalization expended on points that are subsequently clipped

Clip in homogeneous coordinates:

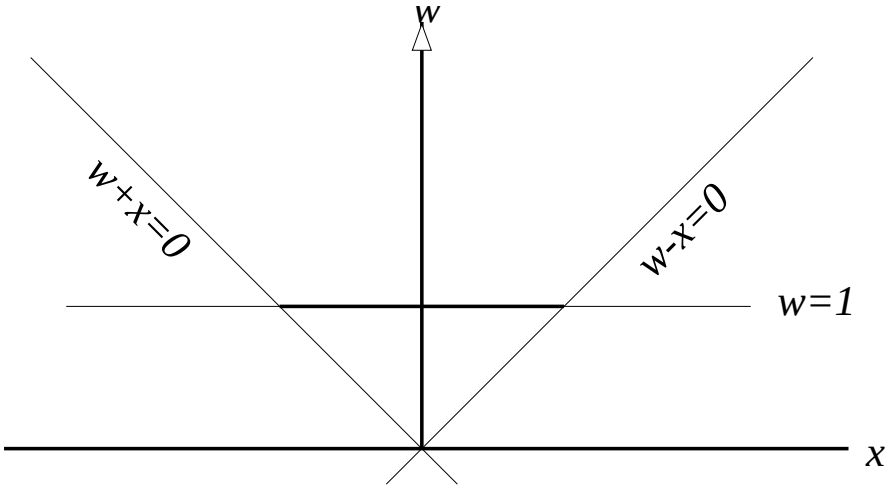
- Compare unnormalized coordinate against $\bar{w}$

$$-|\bar{w}| \leq \bar{x}, \bar{y}, \bar{z} \leq +|\bar{w}|$$

Clipping Homogeneous Coordinates

- Assume NDC window of $[-1, 1] \times [-1, 1]$

- To clip to $X = -1$ (left):
  - Projected coordinates: Clip to $X = -1$
  - Homogeneous coordinate: Clip to $\bar{x}/\bar{w} = -1$
  - Homogeneous plane: $\bar{w} + \bar{x} = 0$



  - Point is visible if $\bar{w} + \bar{x} > 0$

- Repeat for remaining boundaries:
  - $X = \bar{x}/\bar{w} = 1$
  - $Y = \bar{y}/\bar{w} = -1$
  - $Y = \bar{y}/\bar{w} = 1$
  - Near and far clipping planes

9.6 Pinhole Camera vs. Camera vs. Perception

Pinhole Camera vs. Camera

- "Lines map to lines – can't be true because I've seen pictures that curve lines"
- Camera is not a pinhole camera
- Wide angle lens suffers from *barrel distortion*

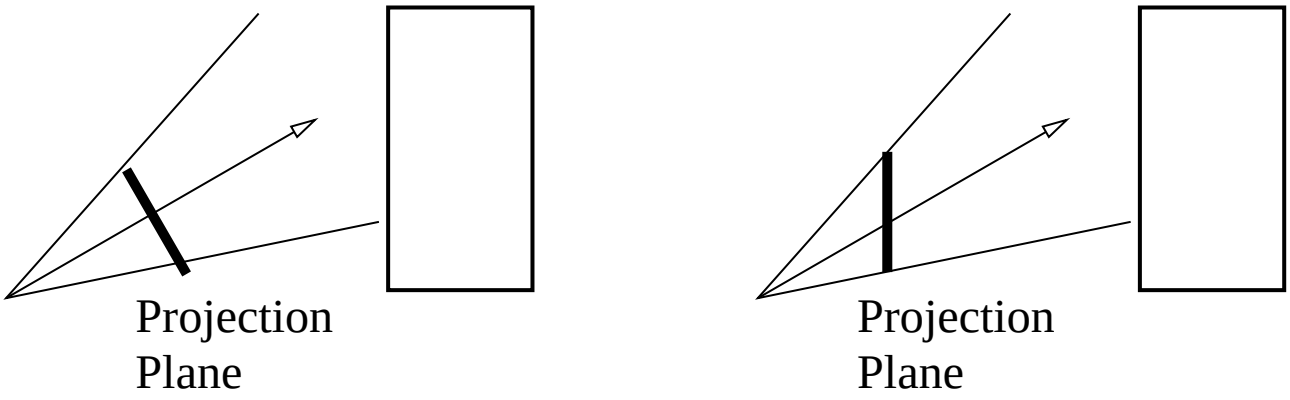
Vertical Convergence

- Is vertical convergence a pinhole camera effect or another kind of distortion?

Slanted apartments

- Vertical convergence is a perspective projection effect.
  Occurs when you tilt the camera/view "up"

- You can get rid of it if view plane parallel to up axis



- Pinhole camera (ray tracer) - Mathematically correct, but looks wrong

  Spheres, cylinders: same size, on a plane parallel to view plane:

Spheres on cylinders

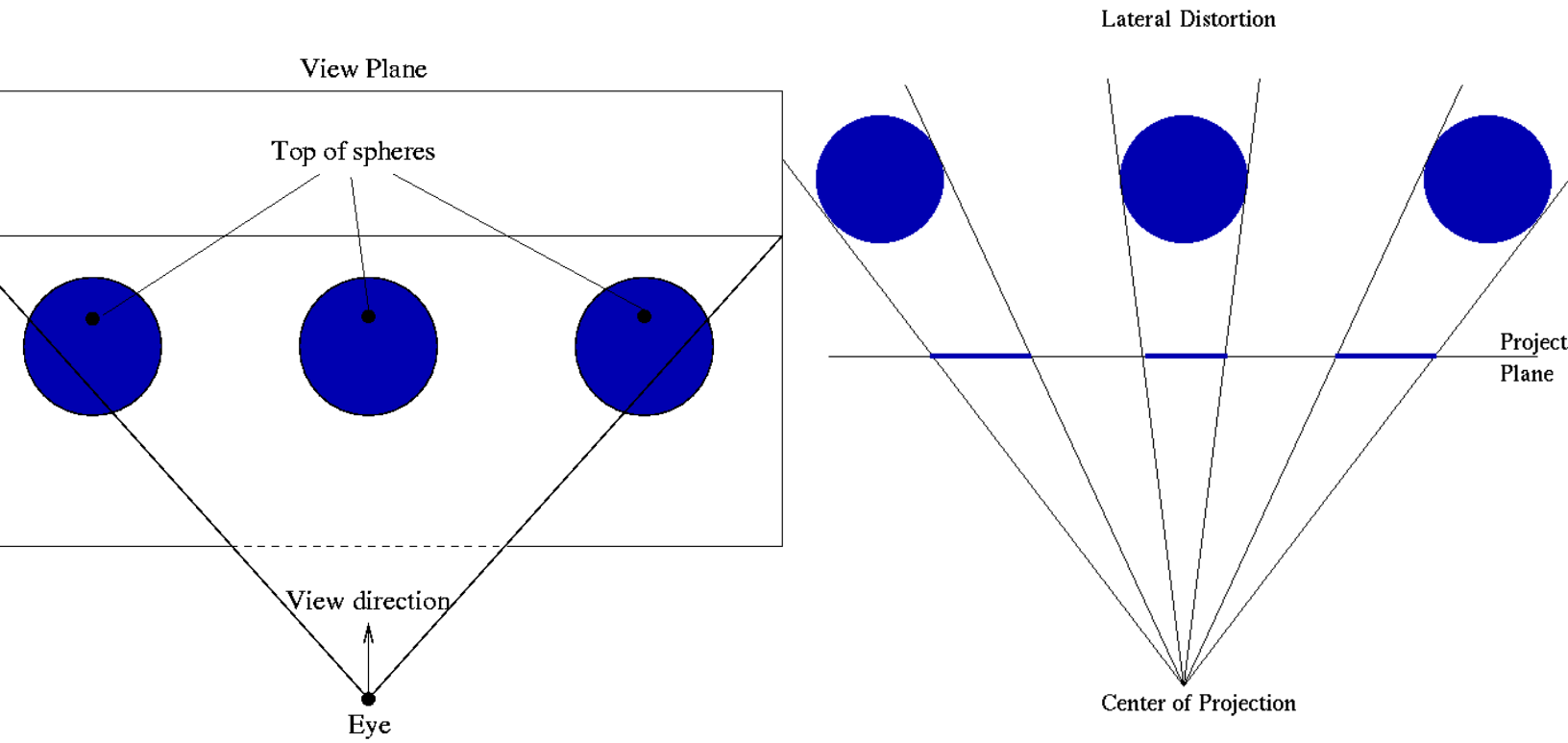
Art

- Painting Sculpture - Mathematically incorrect, but "looks right"

Skew-eyed David

What's going on?

Distortion



Spheres

Spheres, and lots of them!

Occurs in Computer Graphics Images

Ben-hur races in front of distorted columns

In Real Life

- Eye/attention only on small part of field of view

  Sphere looks circular

- Rest of field of view is "there"

  Peripheral spheres not circular, but not focus of attention
  and you don't notice

- When you look at different object, you shift projection plane

  Different sphere looks circular

- In painting, all spheres drawn as circular

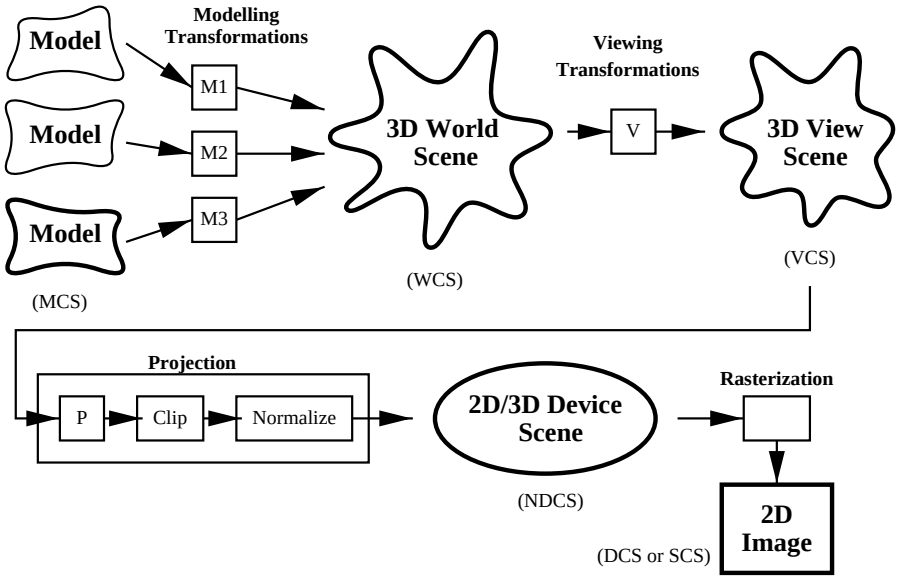
  When not looking at them, they are mathematically wrong
  but since not focus of attention they are "close enough"

- In graphics...

10 Transformation Applications and Extensions

10.1 Rendering Pipeline Revisited

Rendering Pipeline Revisited



Composition of Transforms: $\mathbf{p}' \equiv PVM_i\mathbf{p}$.

10.2 Derivation by Composition

Derivation by Composition

- Can derive the matrix for angle-axis rotation by composing basic transformations.
- Rotation given by $\vec{a} = (x, y, z)$ and θ.
- Assume that $|\vec{a}| = 1$.
- General idea: Map $\vec{a}$ onto one of the canonical axes, rotate by θ, map back.

1. Pick the closest axis to $\vec{a}$ using $\max_i \vec{e}_i \cdot \vec{a} = \max(x, y, z)$. (Assume we chose the x-axis in the following).
2. Project $\vec{a}$ onto $\vec{b}$ in the xz plane:

$$\vec{b} \;=\; (x, 0, z).$$

3. Compute $\cos(\phi)$ and $\sin(\phi)$, where ϕ is the angle of $\vec{b}$ with the x-axis.

$$\begin{aligned} \cos(\phi) &= \frac{x}{\sqrt{x^2 + z^2}}, \\ \sin(\phi) &= \frac{z}{\sqrt{x^2 + z^2}}. \end{aligned}$$

4. Use $\cos(\phi)$ and $\sin(\phi)$ to create $R_y(-\phi)$:

$$R_y(-\phi) = \begin{bmatrix} \cos(-\phi) & 0 & -\sin(-\phi) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(-\phi) & 0 & \cos(-\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

5. Rotate $\vec{a}$ onto the xy plane using $R_y(-\phi)$:

$$\begin{aligned} \vec{c} &= R_y(-\phi)\vec{a} \\ &= \left(\sqrt{x^2+z^2}, y, 0\right). \end{aligned}$$

6. Compute $\cos(\psi)$ and $\sin(\psi)$, where ψ is the angle of $\vec{c}$ with the x-axis.

$$\begin{aligned} \cos(\psi) &= \frac{\sqrt{x^2+z^2}}{\sqrt{x^2+y^2+z^2}} \\ &= \sqrt{x^2+z^2}, \\ \sin(\psi) &= \frac{y}{\sqrt{x^2+y^2+z^2}} \\ &= y. \end{aligned}$$

7. Use $\cos(\psi)$ and $\sin(\psi)$ to create $R_z(-\psi)$:

$$R_z(-\psi) = \begin{bmatrix} \cos(-\psi) & -\sin(-\psi) & 0 & 0 \\ \sin(-\psi) & \cos(-\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

8. Rotate $\vec{c}$ onto the x axis using $R_z(-\psi)$.
9. Rotate about the x-axis by θ: $R_x(-\theta)$.
10. Reverse z-axis rotation: $R_z(\psi)$.
11. Reverse y-axis rotation: $R_y(\phi)$.

The overall transformation is

$$R(\theta, \vec{a}) = R_y(\phi) \circ R_z(\psi) \circ R_x(\theta) \circ R_z(-\psi) \circ R_y(-\phi).$$

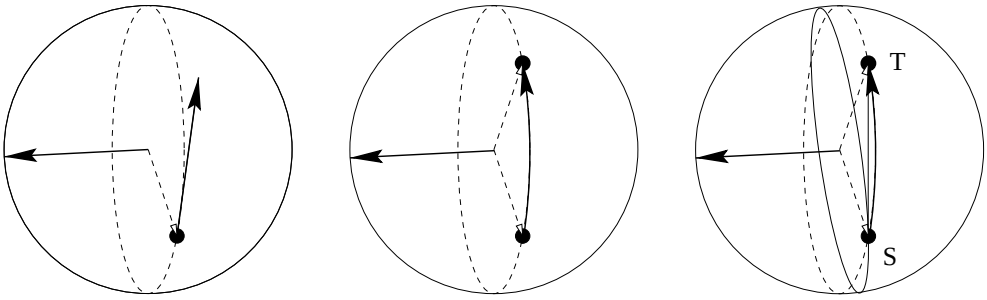
10.3 3D Rotation User Interfaces

3D Rotation User Interfaces

Goal: Want to specify angle-axis rotation "directly".

Problem: May only have mouse, which only has two degrees of freedom.

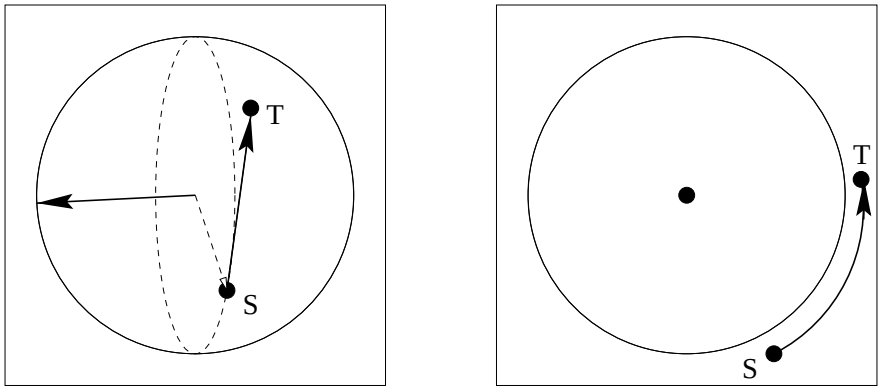
Solutions: Virtual Sphere, Arcball.



Virtual Sphere **Arcball** **Comparison**

10.4 The Virtual Sphere

The Virtual Sphere



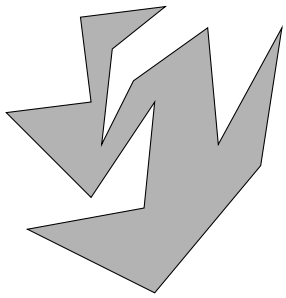
1. Define portion of screen to be projection of virtual sphere.
2. Get two sequential samples of mouse position, S and T.
3. Map 2D point S to 3D unit vector $\vec{p}$ on sphere.
4. Map 2D vector $\vec{ST}$ to 3D tangental velocity $\vec{d}$.
5. Normalize $\vec{d}$.
6. Axis: $\vec{a} = \vec{p} \times \vec{d}$.
7. Angle: $\theta = \alpha|\vec{ST}|$.
   (Choose α so a 180° rotation can be obtained.)
8. Save T to use as S for next time.

11 Polygons

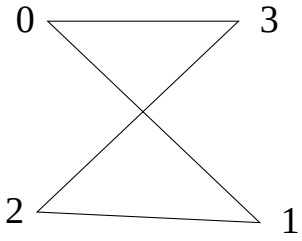
11.1 Polygons – Introduction

Polygons

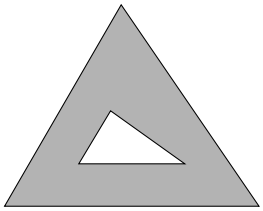
- Need an area primitive
- Simple polygon:
  - Planar set of ordered points, $v_0, \ldots, v_{n-1}$
    (sometimes we repeat v_0 at end of list)
  - No holes
  - No line crossing



OK

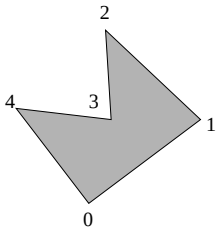


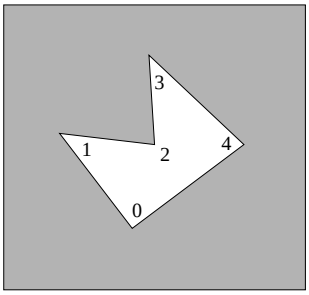
Line Crossing



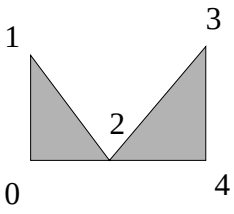
Hole

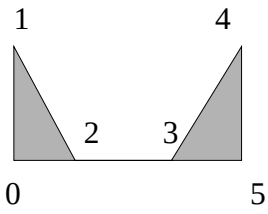
- Normally define an interior and exterior
  Points ordered in counter-clockwise order
  To the "left" as we traverse is inside

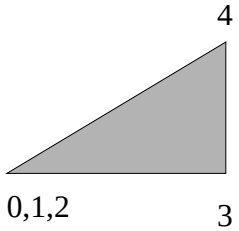




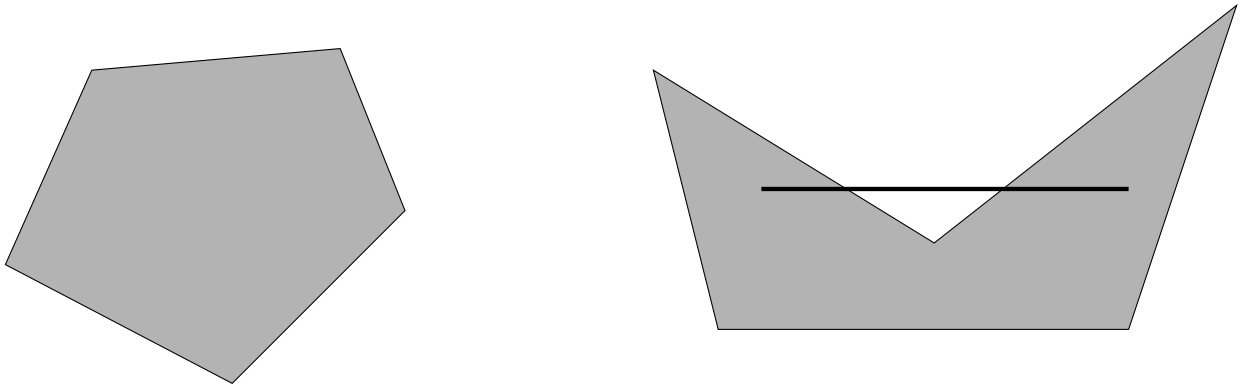
- Try to avoid degeneracies, but sometimes unavoidable







- *Convex* and *Concave* polygons



Polygon is convex if for any two points inside polygon, the line segment joining these two points is also inside.

Convex polygons behave better in many operations

- Affine transformations may introduce degeneracies

  Example: Orthographic projection may project entire polygon to a line segment.

11.2 Polygon Clipping

Polygon Clipping (Sutherland-Hodgman):

- Window must be a convex polygon
- Polygon to be clipped can be convex or not

Approach:

- Polygon to be clipped is given as $v_1, \ldots, v_n$
- Each polygon edge is a pair $[v_i, v_{i+1}]$ $i = 1, \ldots, n$
  - Don't forget wraparound; $[v_n, v_1]$ is also an edge
- Process all polygon edges in succession against a window edge
  - Polygon in – polygon out
  - $v_1, \ldots, v_n \rightarrow w_1, \ldots, w_m$
- Repeat on resulting polygon with next sequential window edge

Contrast with Line Clipping:

- **Line Clipping:**
  - Clip only against possibly intersecting window edges
  - Deal with window edges in any order
  - Deal with line segment endpoints in either order
- **Polygon Clipping:**
  - Each window edge must be used

- Polygon edges must be handled in sequence
- Polygon edge endpoints have a given order
- Stripped-down line-segment/window-edge clip is a subtask

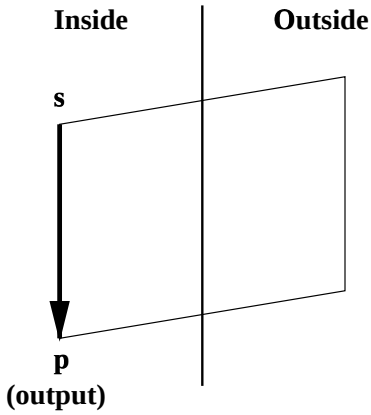
Notation:

- $\mathbf{s} = v_i$ is the polygon edge starting vertex
- $\mathbf{p} = v_{i+1}$ is the polygon edge ending vertex
- **i** is a polygon-edge/window-edge intersection point
- w_j is the next polygon vertex to be output

There are four cases to consider

Case 1: Polygon edge is entirely inside the window edge

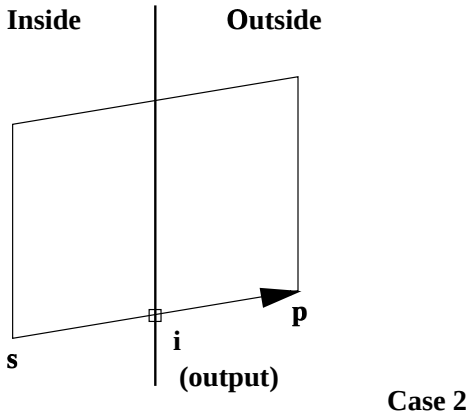
- **p** is next vertex of resulting polygon
- $\mathbf{p} \rightarrow w_j$ and $j + 1 \rightarrow j$



Case 1

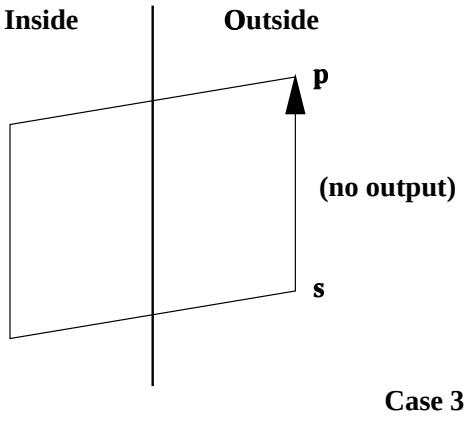
Case 2: Polygon edge crosses window edge going out

- Intersection point **i** is next vertex of resulting polygon
- $\mathbf{i} \rightarrow w_j$ and $j + 1 \rightarrow j$



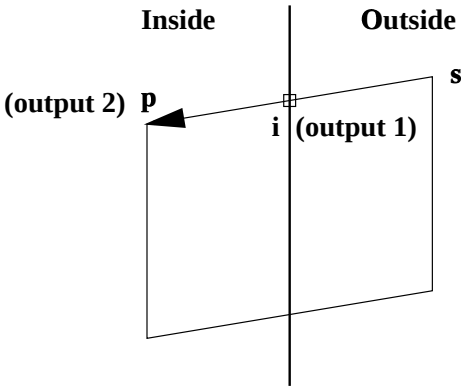
Case 3: Polygon edge is entirely outside the window edge

- No output



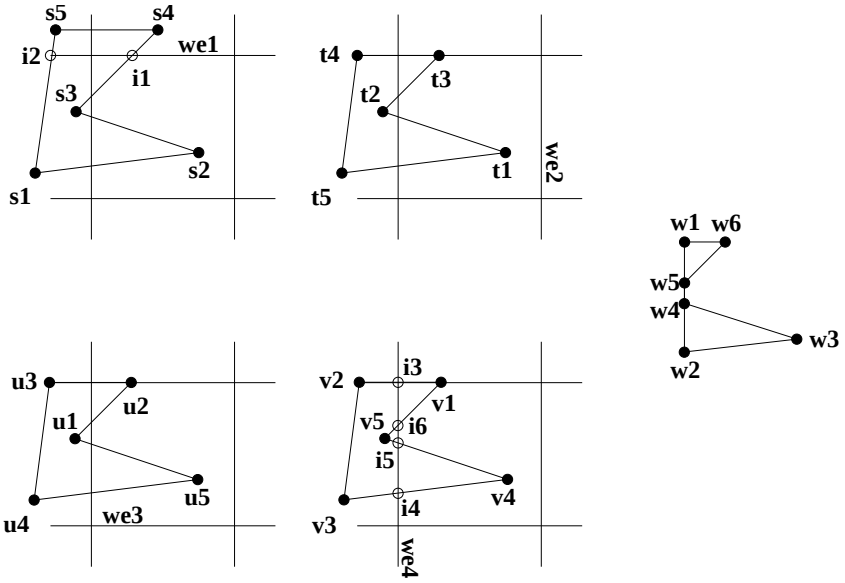
Case 4: Polygon edge crosses window edge going in

- Intersection point **i** and **p** are next two vertices of resulting polygon
- **i** $\rightarrow w_j$ and **p** $\rightarrow w_{j+1}$ and $j + 2 \rightarrow j$



Case 4

An Example: With a non-convex polygon...

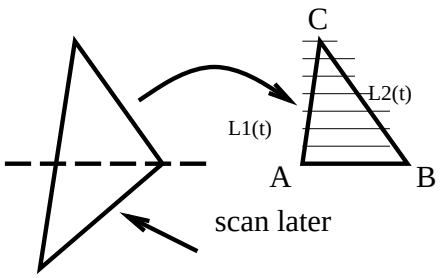


(Readings: Watt: 6.1. Hearn and Baker: Section 6-8. Red book: 3.11. White book: 3.14.)

11.3 Polygon Scan Conversion

Scan Conversion

- Once mapped to device coordinates, want to scan convert polygon.
- Scan converting a general polygon is complicated.
- Here we will look at scan conversion of a triangle.
- Look at y value of vertices. Split triangle along horizontal line at middle y value.



- Step along $L1$ and $L2$ together along the scan lines from A to C and from B to C respectively.
- Scan convert each horizontal line.

Code for triangle scan conversion

- Assume that triangle has been split and that A, B, C are in device coordinates, that $A.x < B.x$, and that $A.y = B.y \neq C.y$.

```
y = A.y;
d0 = (C.x-A.x)/(C.y-A.y);
d1 = (C.x-B.x)/(C.y-B.y);
x0 = A.x;
x1 = B.x;
while ( y <= C.y ) do
   for ( x = x0 to x1 ) do
       WritePixel(x,y);
   end
   x0 += d0;
   x1 += d1;
   y++;
end
```

- This is a floating point algorithm

(Readings: Watt: 6.4. Hearn and Baker: Chapter 3-11. Red book: 3.5 (more general than above). White book: 3.6 (more general than above).)

11.4 Dos and Don'ts

When modelling with polygonal objects, remember the following guidelines:

- Model Solid Objects — No façades

  It's easier to write other software if we can assume polygon faces are oriented

- No T-vertices

  These cause shading anomalies and pixel dropout

- No overlapping co-planar faces

  Violates solid object constraint.

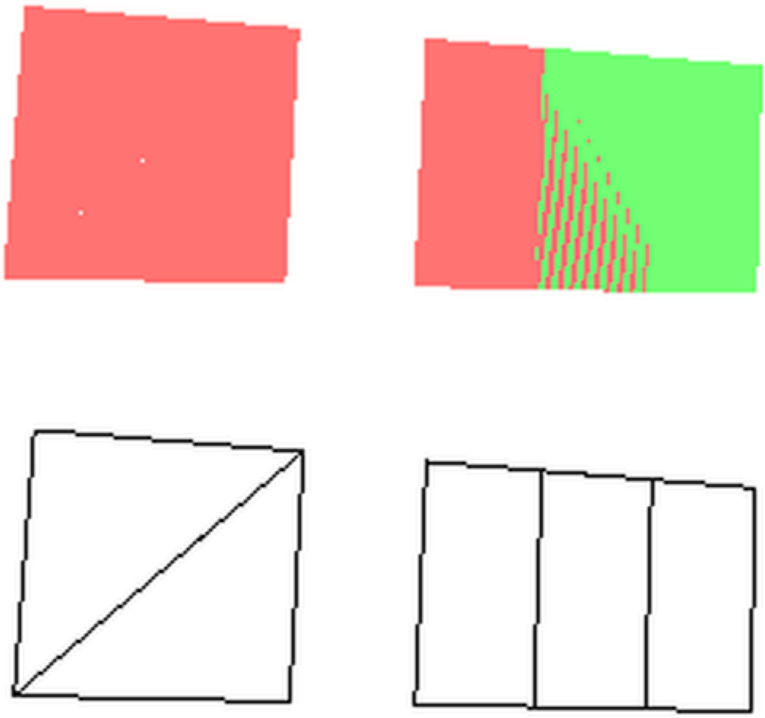
  If different colours, then psychedelic effect due to floating point roundoff

- Well shaped polygon

  Equilateral triangle is best.

  Long, skinny triangles pose problems for numerical algorithms

Bad Polygon Examples



12 Hidden Surface Removal

12.1 Hidden Surface Removal

Hidden Surface Removal

- When drawing lots of polygons, we want to draw only those "visible" to viewer.
- There are a variety of algorithms with different strong points.
- Issues:
  - Online
  - Device independent
  - Fast
  - Memory requirements
  - Easy to implement in hardware

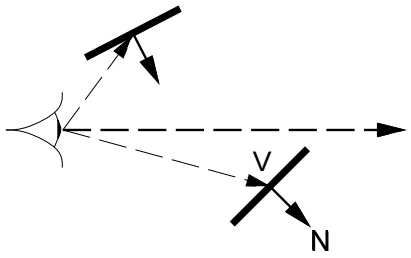
(Readings: Watt: 6.6. Red book: 13. White book: 15. Hearn and Baker: Chapter 13.)

12.2 Backface Culling

Backface Culling

Backface Culling

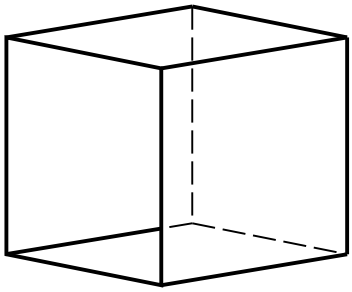
- A simple way to perform hidden surface is to remove all "backfacing" polygons.
- If polygon normal facing away from the viewer then it is "backfacing."
- For solid objects, this means the polygon will not be seen by the viewer.



- Thus, if $N \cdot V > 0$, then cull polygon.
- Note that V is vector from eye to point on polygon
  You cannot use the view direction for this.

Backface Culling Not a complete solution

- If objects not convex, need to do more work.
- If polygons two sided (i.e., they do not enclose a volume) then we can't use it.

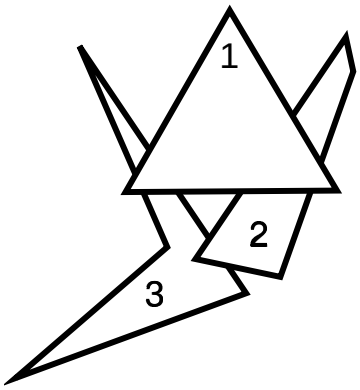


- A HUGE speed advantage if we can use it since the test is cheap and we expect at least half the polygons will be discarded.
- Usually performed in conjunction with a more complete hidden surface algorithm.
- Easy to integrate into hardware (and usually improves performance by a factor of 2).

12.3 Painter's Algorithm

Painter's Algorithm

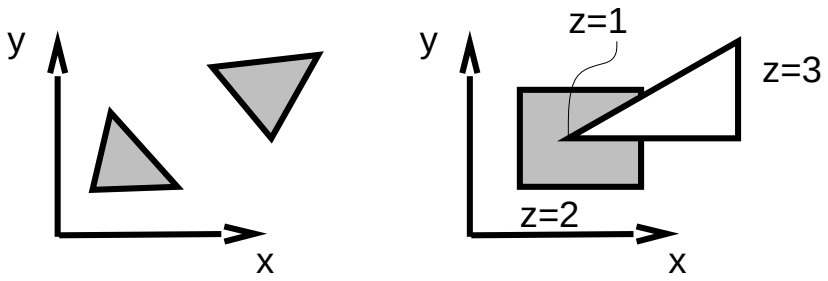
- Idea: Draw polygons as an oil painter might: The farthest one first.
  - Sort polygons on farthest z
  - Resolve ambiguities where z's overlap
  - Scan convert from largest z to smallest z



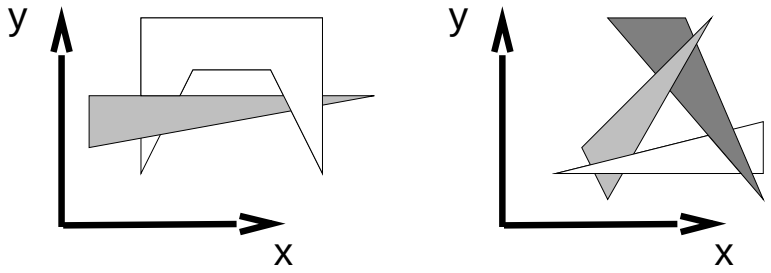
- Since closest drawn last, it will be on top (and therefore it will be seen).
- Need all polygons at once in order to sort.

Painter's Algorithm — Z overlap

- Some cases are easy:



- But other cases are nasty!



(have to split polygons)

- $\Omega(n^2)$ algorithm, lots of subtle detail

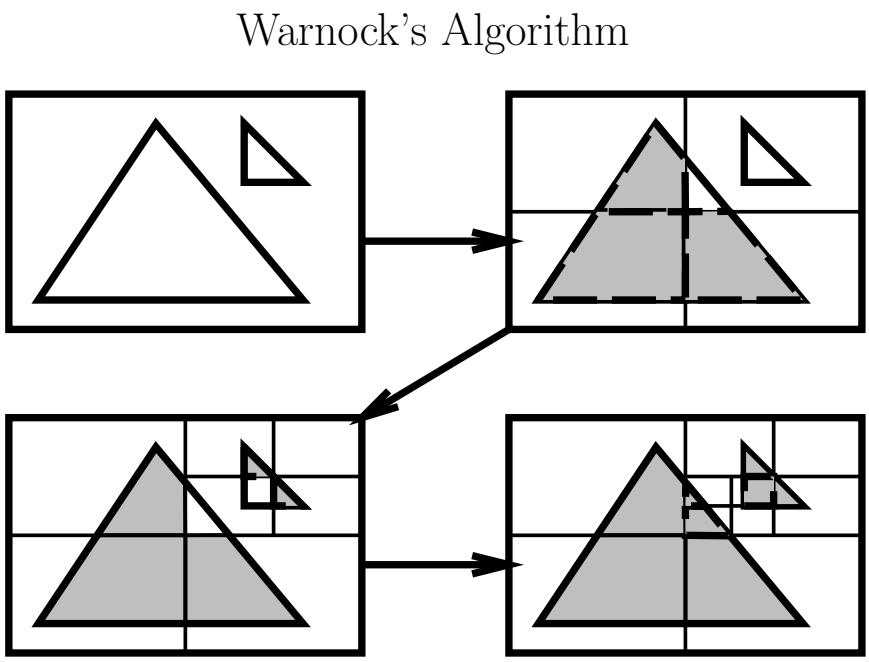
12.4 Warnock's Algorithm

Warnock's Algorithm

- A divide and conquer algorithm

```
Warnock(PolyList PL, ViewPort VP)
If ( PL simple in VP) then
    Draw PL in VP
else
    Split VP vertically and horizontally into VP1,VP2,VP3,VP4
    Warnock(PL in VP1, VP1)
    Warnock(PL in VP2, VP2)
    Warnock(PL in VP3, VP3)
    Warnock(PL in VP4, VP4)
end
```

- What does "simple" mean?
  - No more than one polygon in viewport
    Scan convert polygon clipped to viewport
  - Viewport only 1 pixel in size
    Shade pixel based on closest polygon in the pixel



- Runtime: $O(p \times n)$
  - p: number of pixels
  - n: number of polygons

12.5 Z-Buffer Algorithm

Z-Buffer Algorithm

- Perspective transformation maps viewing pyramid to viewing box in a manner that maps lines to lines
- This transformation also maps polygons to polygons
- Idea: When we scan convert, step in z as well as x and y.
- In addition to framebuffer, we'll have a depth buffer (or z buffer) where we write z values
- Initially, z buffer values set to ∞

  Depth of far clipping plane (usually 1) will also suffice
- Step in z, both in the while loop and in the for loop.
- Scan convert using the following WritePixel:

```
WritePixel(int x, int y, float z, colour)
if ( z < zbuf[x][y] ) then
    zbuf[x][y] = z;
    frambuffer[x][y] = colour;
end
```

- Runtime: $O(p_c + n)$
  - p_c: number of scan converted pixels
  - n: number of polygons

Z-buffer in Hardware

Z-buffer is the algorithm of choice for hardware implementation

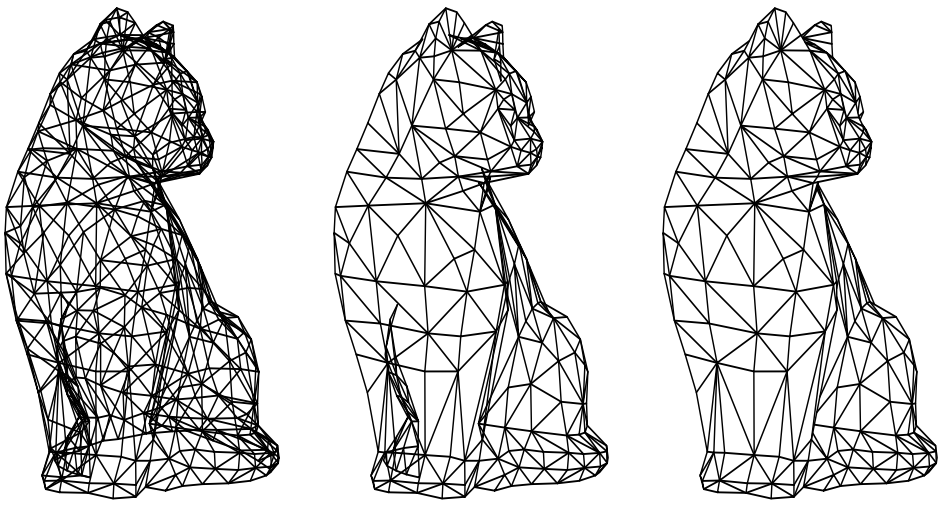
+ Easy to implement
+ Simple hardware implementation
+ Online algorithm (i.e., we don't need to load all polygons at once in order to run algorithm)
- Doubles memory requirements (at least)
  But memory is cheap!
- Scale/device dependent

12.6 Comparison of Algorithms

Comparison of Algorithms

- Backface culling fast, but insufficient by itself
- Painter's algorithm device independent, but details tough, and algorithm is slow
- Warnock's algorithm easy to implement, but not very fast, and is semi-device dependent.
- Z-buffer online, fast, and easy to implement, but device dependent and memory intensive. Algorithm of choice for hardware

Comparison of no hidden surface removal, backface culling, and hidden surface removal:



13 Hierarchical Models and Transformations

13.1 Hierarchical Transformations

Hierarchical Transformations

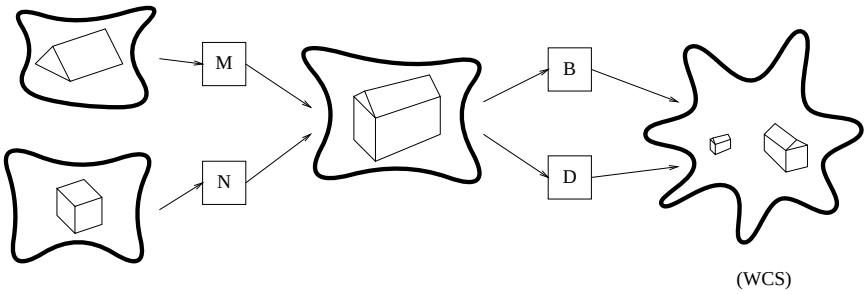
How do we model complex objects and scenes?

- Describing everything as a single complex model is a Bad Idea.
- Use hierarchical modeling instead...

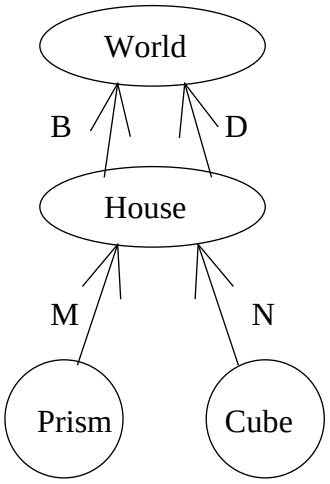
Start with basic set of 3D primitives: Cubes, spheres, prisms ...

- Each is defined in a "nice" way in its own space.
- To put primitives together, use transformations.
- Each transformation embeds one space in another.
- Use a whole hierarchy of spaces to build up complex models...
- Not just one "Model space", but one for each model and part of a model.

Suppose we want two houses.



Pictorially we have a DAG —a directed acyclic graph.



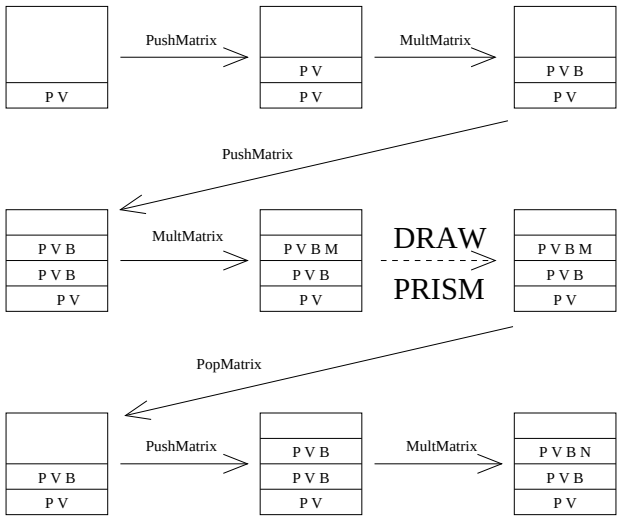
We can model this procedurally:

```
Procedure Scene()
    House(B);
    House(D);
end

Procedure House(E)
    Prism(E o M);
    Cube(E o N);
end

Procedure Cube(F)
    . . .
end

Procedure Prism(F)
    . . .
end
```

Implementation:

- Procedure calls are making depth first traversal of tree/DAG.
- Use a **matrix stack**—sometimes maintained in H/W.
- Each time we make an embedding,
  1. "Push" the new transformation onto the matrix stack
  2. "Pop" it on return.
- OpenGL's `glPushMatrix` call duplicates the top of the stack.
- OpenGL's `glMultMatrix` multiplies a matrix with the top of the stack, replacing the top of the stack with the result.
- OpenGL's `glPopMatrix` pops the stack.

- Stack starts with identity matrix on bottom.
- OpenGL also has transformation calls, such as `glRotate`, `glScale`, etc that perform basic transformations.
- These transformations operate on the top of the stack.
- Put perspective and world-to-view matrices into the stack.
- These are pushed on first, giving PV on the bottom.
- Might have more than one stack, i.e. MODEL, VIEW, and PROJECTION stacks.



• • •

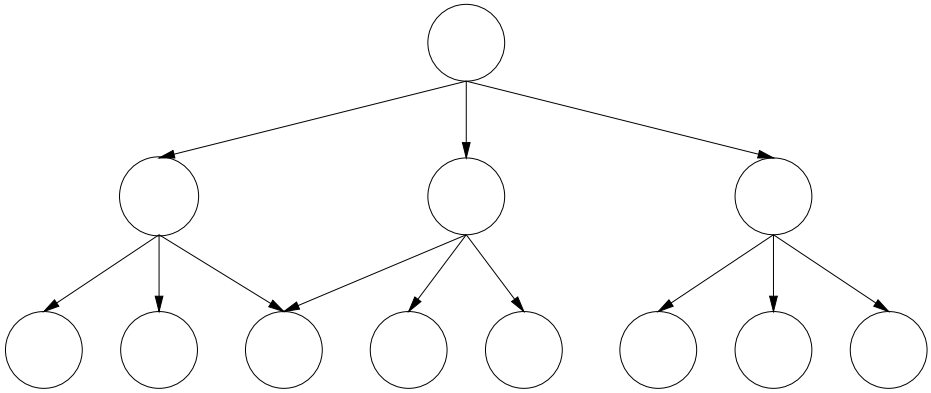
Code now looks like

```
Procedure Scene()
    MultMatrix(P);
    MultMatrix(V);
    PushMatrix();
        MultMatrix(B);
        House();
    PopMatrix();
    PushMatrix();
        MultMatrix(D);
        House();
    PopMatrix();
end

Procedure House()
    PushMatrix();
        MultMatrix(M);
        Prism();
    Popmatrix();
    PushMatrix();
        MultMatrix(N);
        Cube();
    Popmatrix();
end

Procedure Cube()        Procedure Prism()
    . . .                   . . .
end                     end
```

13.2 Hierarchical Data Structures

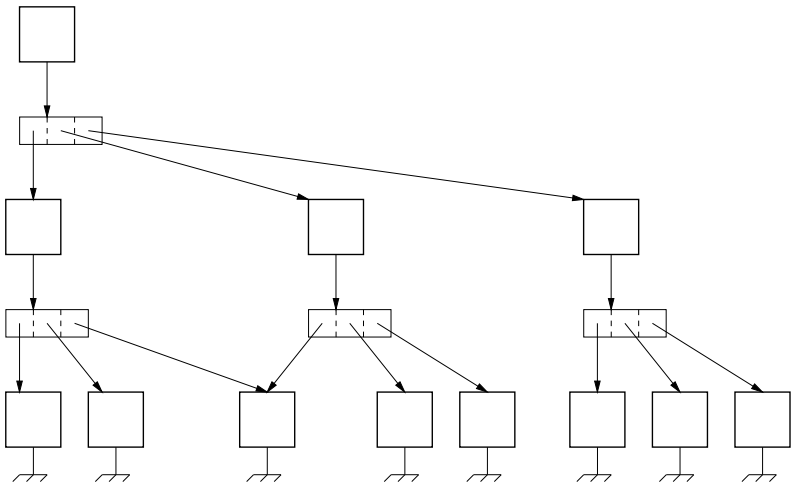
Hierarchical Data Structures

- Don't want to write a program for each scene...
- Instead, develop a data structure and traverse it.
- Use an external data format to describe model hierarchies.
- Process many different scenes with the same program.
- DAG data structure: lots of variations on the example given here.
- Consider the following DAG:



- Use one of a variety of data structures to allow an arbitrary number of links from each node. Example:

```
class Node {
  ...
  private:
    Primitive *prim;
    vector<Node*> children;
};
```



- We walk through the hierarchy with a preorder traversal...
- Traversal pseudocode:

```
void Node::traverse() {
    PushMatrix(); MultMatrix(transform);
      if (prim) {
            prim->draw();
        } else {
            for (vector<Node*> iterator i=children.begin();
                 i != children.end(); ++i) {
              (*i)->traverse();
            }
        }
    PopMatrix();
}
```

A3/A4 House Code

```
scene = gr.transform( 'scene' )

# Assume we have predefined materials 'red', 'green' and 'blue'

house = gr.transform( 'house' )
prism = gr.polyhedron( 'prism', ... )

roof = gr.transform( 'roof' )
gr.add_child( roof, prism )
gr.set_transform( roof, gr.translation( 0,1,0 ) )
gr.add_child( house, roof )
frame = gr.cube( 'frame' )
gr.add_child( house, frame )

farmhouse = gr.transform( 'farmhouse' )
gr.add_child( farmhouse, house )
gr.set_material( farmhouse, green )
```

(continued on next slide)
(continuation from previous slide)

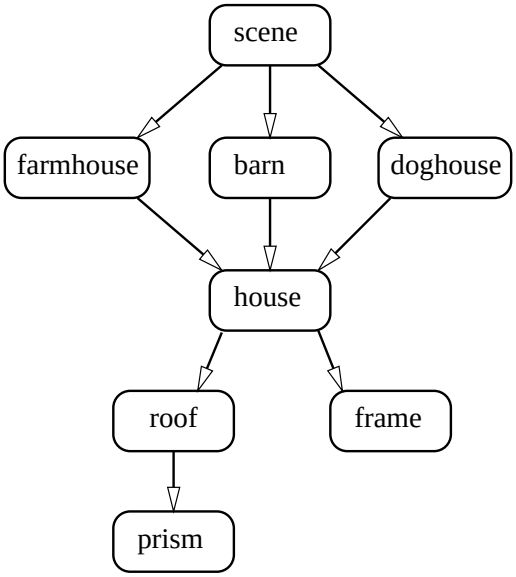
```
barn = gr.transform( 'barn' )
gr.add_child( barn, house )
gr.set_material( barn, red )
gr.set_transform( barn, gr.translation( 2,0,3 ) * gr.scaling( 2,2,2 ) \
        * gr.rotation( 'Y', 30 ) )

doghouse = gr.transform( 'doghouse' )
gr.add_child( doghouse, house )
gr.set_material( doghouse, blue )
gr.set_transform( doghouse, gr.translation( 1.5, 0, -0.2 ) \
        * gr.scaling( 0.2,0.2,0.2 ) * gr.rotation( 'Y', -15 ) )

gr.add_child( scene, farmhouse )
```

```
gr.add_child( scene, barn )
gr.add_child( scene, doghouse )

# Later in the code...
... gr.render( scene ) ...
```

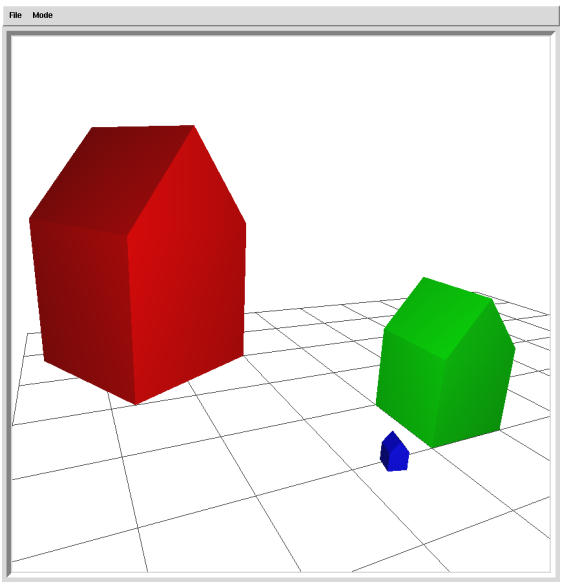
Improved A3/A4 House Code

```
# Assume materials are defined
# About 30 lines defining make_ functions left out here.

# Build a group node whose children are the arguments to the function.
house = make_group(
    make_cube(), make_poly( prism_data, transform=gr.translation( 0,1,0 ) )
)

farmhouse = make_instance( house, material=green )
barn = make_instance( house, material=red,
    transform=gr.translation(2,0,3)*gr.scaling(2,2,2)*gr.rotation('Y',30) )
doghouse = make_instance( house, material=blue,
    transform=gr.translation(1.5,0,-0.2)*gr.scaling(0.2,0.2,0.2)*
         gr.rotation('Y',-15) )

scene = make_group( farmhouse, barn, doghouse )

... gr.render( scene ) ...
```

- Note: Conceptually, DAG is



- Primitives only occur at leaf nodes.
- Transformations only occur at internal nodes.
- Rendered scene:



- How to associate materials with objects?
- For assignments, just associate with primitives.
- More complex:
  - Material assigned to inner nodes
    Who has precedence, lower or higher in hierarchy?
  - Parameterized materials
    - Subtree with materials
      (house with white paint, green roof)
    - Might want instances of house to have different materials
      (cream paint, black roof)
    - Have to parameterize materials on creation
    - Have to have method for setting materials in subtree

14 Picking and 3D Selection

14.1 Picking and 3D Selection

Picking and 3D Selection

- **Pick:** Select an object by positioning mouse over it and clicking.
- **Question:** How do we decide what was picked?
  - We could do the work ourselves:
    - Map selection point to a ray;
    - Intersect with all objects in scene.
  - Let OpenGL/graphics hardware do the work.
- **Idea:** Draw entire scene, and "pick" anything drawn near the cursor.
  - Only "draw" in a small viewport near the cursor.
  - Just do clipping, no shading or rasterization.
  - Need a method of identifying "hits".
  - OpenGL uses a **name stack** managed by `glInitNames()`, `glLoadName()`, `glPushName()`, and `glPopName()`.
  - "Names" are unsigned integers
  - When hit occurs, copy entire contents of stack to output buffer.
- Process:
  - Set up pick buffer
  - Initialize name stack
    May need `glPushName(-1);`
  - Save old projective transformation and set up a new one
  - Draw your scene with appropriate use of name stack
  - Restore projective transformation
  - Query for number of hits and process them
- A hit occurs if a draw occurs in the (small) viewport
- All information (except 'hits') is stored in buffer given with `glSelectBuffer`
- **Example:**
  - Two objects
  - Might pick none
  - Might pick either one
  - Might pick both

```
glSelectBuffer(size, buffer);      /* initialize */
glRenderMode(GL_SELECT);
glInitNames();
glPushName(-1);
glGetIntegerv(GL_VIEWPORT,viewport);    /* set up pick view */
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(x,y,w,h,viewport);
glMatrixMode(GL_MODELVIEW);

ViewMatrix();
glLoadName(1);
Draw1();
glLoadName(2);
Draw2();

glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
hits = glRenderMode(GL_RENDER);
```

- Hits stored consecutively in array
- In general, if h is the number of hits, the following is returned.
  - `hits` = h.
  - h **hit records**, each with four parts:
    1. The number of items q on the name stack at the time of the hit (1 int).
    2. The minimum z value among the primitives hit (1 int).
    3. The maximum z value among the primitives hit (1 int).
    4. The contents of the hit stack, deepest element first (q ints).
- At most one hit between each call `glRenderMode` or a change to name stack
- In our example, you get back the following:
  - If you click on Item 1 only:
    `hits = 1`,
    `buffer = 1, min(z1), max(z1), 1`.
  - If you click on Item 2 only:
    `hits = 1`,
    `buffer = 1, min(z2), max(z2), 2`.
  - If you click over both Item 1 and Item 2:
    `hits = 2`,
    `buffer = 1, min(z1), max(z1), 1,`
    `1, min(z2), max(z2), 2`.
- **More complex example:**

```
/* initialization stuff goes here */
glPushName(1);
    Draw1();        /* stack: 1 */
    glPushName(1);
        Draw1_1();      /* stack: 1 1 */
        glPushName(1);
            Draw1_1_1();    /* stack: 1 1 1 */
        glPopName();
        glPushName(2);
            Draw1_1_2();    /* stack: 1 1 2 */
        glPopName();
    glPopName();
    glPushName(2);  /* stack: 1 2 */
        Draw1_2();
    glPopName();
glPopName();
glPushName(2);
    Draw2();        /* stack: 2 */
glPopName();
/* wrap-up stuff here */
```

- What you get back:
  - If you click on Item 1:
    `hits = 1`,
    `buffer = 1, min(z1), max(z1), 1`.
  - If you click on Items 1:1:1 and 1:2:
    `hits = 2`,
    `buffer = 3, min(z111), max(z111), 1, 1, 1,`
    `2, min(z12), max(z12), 1, 2`.
  - If you click on Items 1:1:2, 1:2, and 2:
    `hits = 3`,
    `buffer = 3, min(z112), max(z112), 1, 1, 2,`
    `2, min(z12), max(z12), 1, 2,`
    `1, min(z2), max(z2), 2`.
- Important Details:
  - Make sure that projection matrix is saved with a `glPushMatrix()` and restored with a `glPopMatrix()`.
  - `glRenderMode(GL_RENDER)` returns negative if buffer not big enough.
  - When a hit occurs, a flag is set.
  - Entry to name stack only made at next `gl*Name(s)` or `glRenderMode` call. So, each draw block can only generate at most one hit.

15 Colour and the Human Visual System

15.1 Colour

Introduction to Colour

- Light sources emit intensity:

  $$I(\lambda)$$

  assigns intensity to each wavelength of light
- Humans perceive $I(\lambda)$ as a colour – navy blue, light green, etc.
- Experiments show that there are distinct I perceived as the same colour (metamers)
- Normal human retina have three types of colour receptors which respond most strongly to short, medium, or long wavelengths.

Sensitivity diagram

- Note the low response to blue.

  One theory is that sensitivity to blue is recently evolved.
- Different animals have different number of wavelengths that they are sensitive to:
  - Dogs: 1
  - Primates: 2 or 3
  - Pigeon: 4
  - Birds: up to 18 (hummingbird?)
- Different Regions of the eye are "designed" for different purposes:
  - Center - fine grain colour vision
  - Sides - night vision and motion detection

Tri-Stimulus Colour Theory

- Tri-stimulus Colour Theory models visual system as a linear map:

  $$V : \Lambda \to C$$

  where Λ is a continuous function of wavelength

  C is a three dimensional vector space (colour space)

  $$V(I(\lambda)) = \vec{C} = \ell\vec{\ell} + m\vec{m} + s\vec{s}$$

  where

  $$\begin{aligned} \ell &= \int_0^\infty L(\lambda)I(\lambda)d\lambda \\ m &= \int_0^\infty M(\lambda)I(\lambda)d\lambda \\ s &= \int_0^\infty S(\lambda)I(\lambda)d\lambda \end{aligned}$$

  where L, M, and S are weight functions.

Colour Systems

- RGB (Red, Green, Blue) Additive

Prism to RBG/CMY wheel

- CMY (Cyan, Magenta, Yellow) Subtractive (complement of RGB)

Additive colour scheme

  Often add K (blacK) to get better black

- HSV (Hue, Saturation, Value) Cone shaped colour space

HSV colour cone

  Also HSL (double cone)

- CIE XYZ (Colour by committee) More complete colour space
  Also L*u*v and L*a*b

- YIQ (Y == Luminance == CIE Y; IQ encode colour)
  Backwards compatible with black-and-white TV (only show Y)

16 Reflection and Light Source Models

16.1 Goals

Lighting

- What we want:

  Given a point on a surface visible to the viewer through a pixel, what colour should we assign to the pixel?

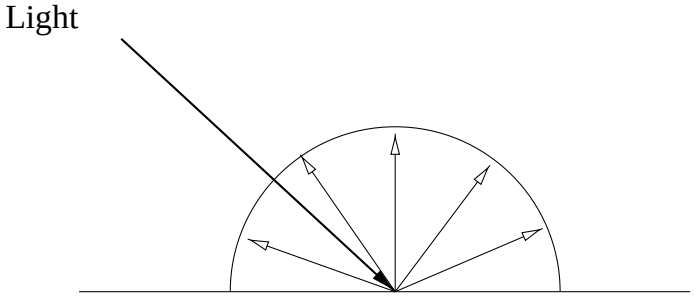
- Goals:
  - Smoothly shade objects in scene
  - Want shading done quickly
    Interactive speeds
- Begin with a simple lighting model at a single point on a surface
- Not going to worry *too* much about the physics...
- Later, we will improve lighting model
- Initial Assumptions:
  - Linearity of reflection: outgoing energy proportional to incoming
  - Energy conservation: no more outgoing energy than incoming
  - Full spectrum of light can be represented by three floats (Red,Green,Blue)

(Readings: Watt: 6.2. Red book: 14.1. White book: 16.1.)

16.2 Lambertian Reflection

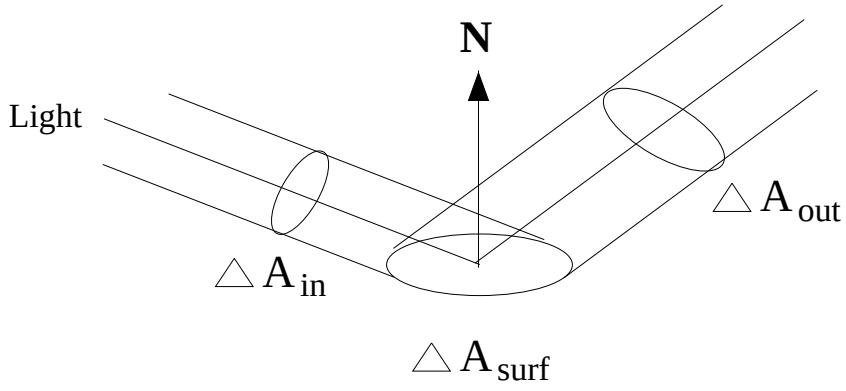
Lambertian Reflection

- Assume: Incoming light is partially absorbed, then remainder of energy propagated **equally** in all directions



- Approximates the behavior of matte materials.

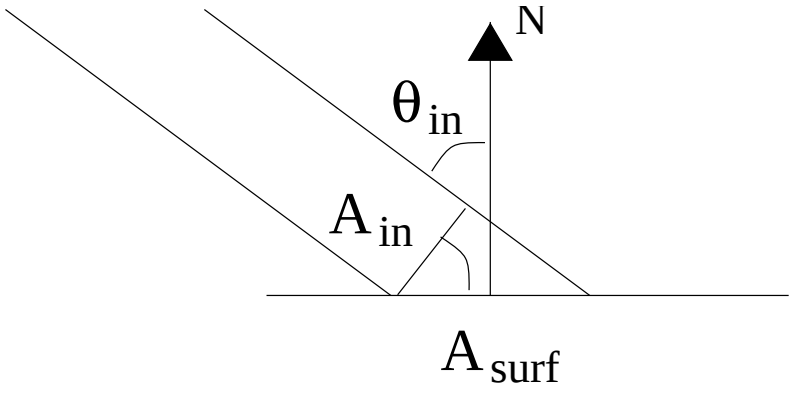
- Want an expression for $L_{\text{out}}(\vec{v})$, the radiance (light energy transported along a ray) reflected in some direction $\vec{v}$, given
  - the incoming direction $\vec{\ell}$
  - incoming radiance $L_{\text{in}}(\vec{\ell})$,
  - and the material properties (reflectance).
- Given radiance $L_{\text{in}}(\vec{\ell})$ striking surface from direction $\vec{\ell}$
- Want to determine $L_{\text{out}}(\vec{v})$ reflected in direction $\vec{v}$.



- Energy received at surface (irradiance E) depends on projection of incoming beam orientation onto surface's orientation:

$$E_{\text{in}}(\vec{\ell}) \quad = \quad L_{\text{in}}(\vec{\ell}) \left(\frac{\Delta A_{\text{in}}}{\Delta A_{\text{surf}}} \right).$$

- Want to compute projected area:



$$\begin{aligned} \frac{\Delta A_{\text{in}}}{\Delta A_{\text{surf}}} &= \cos(\theta_{\text{in}}) \\ &= \vec{\ell} \cdot \vec{n} \end{aligned}$$

where $\vec{n}$ is the surface normal and $\vec{\ell}$ points *towards* the light.

- Our "view" of the surface will include a corresponding factor for $\cos(\theta_{\text{out}})$.
- Outgoing radiance proportional to incoming irradiance, but proportionality may depend on view vector $\vec{v}$ and light vector $\vec{\ell}$:

$$L_{\text{out}}(\vec{v}) = \rho(\vec{v}, \vec{\ell}) E_{\text{in}}(\vec{\ell}).$$

- Have assumed outgoing radiance is equal in all directions so ρ must be a constant.
- The *Lambertian lighting model* is therefore

$$\begin{aligned} L_{\text{out}}(\vec{v}) &= k_d E_{\text{in}}(\vec{\ell}) \\ &= k_d L_{\text{in}}(\vec{\ell})\, \vec{\ell} \cdot \vec{n}. \end{aligned}$$

- For complete environment, Lambertian lighting model is

$$L_{\text{out}}(\vec{v}) = \int_{\Omega} (k_d/\pi) L_{\text{in}}(\vec{\ell})\ (\vec{\ell} \cdot \vec{n})\ d\sigma(\vec{\ell})$$

  where Ω is the hemisphere of all possible incoming directions and $d\sigma$ is the solid angle measure.
- If $k_d \in [0, 1]$, then factor of π is necessary to ensure conservation of energy.
- Often convert integral to a sum over light sources

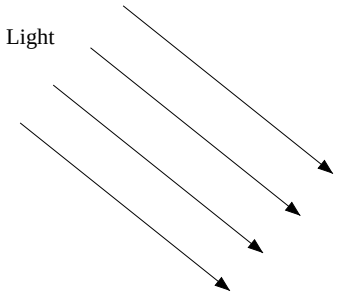
16.3 Attenuation

Attenuation

Attenuation

- We will model two types of lights:
  - Directional
  - Point

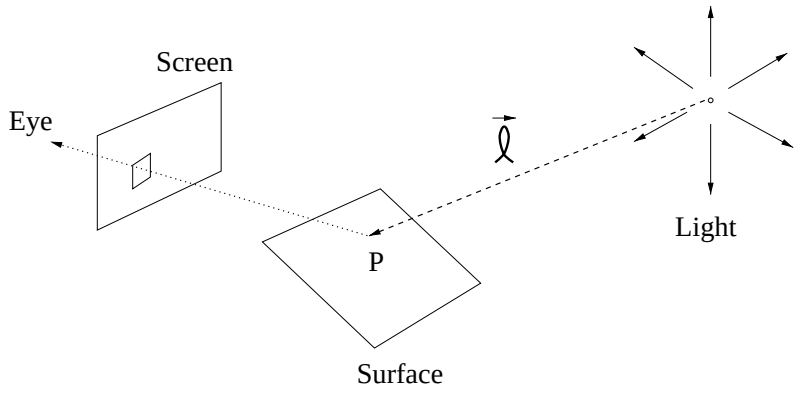
  Want to determine $L_{\text{in}}(\vec{\ell})$ at surface for each
- Directional light source has parallel rays:



Most appropriate for distant light sources (the sun)
No attenuation, since energy does not "spread out".

Point light sources:

- Light emitted from a point equally in all directions:



- Conservation of energy tells us

$$L_{\text{in}}(\vec{\ell}) \propto \frac{I}{r^2}$$

  where r is the distance from light to P (energy is spread over surface of a sphere), and I is light source intensity.

- For "empirical graphics", usually ignore π factors...

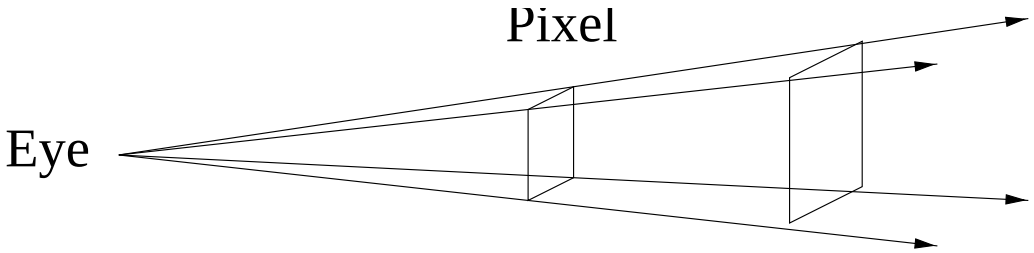
- However, r^2 attenuation looks too harsh.

  Harshness because real lighting is from area sources, multiple reflections in environment. True point sources impossible.

  Commonly, we will use

$$L_{\text{in}}(\vec{\ell}) = \frac{I}{c_1 + c_2 r + c_3 r^2}$$

- Note that we do NOT attenuate light from P to screen. Surface foreshortening when farther away takes care of that:



  The pixel represents an area that increases as the square of the distance.

16.4 Coloured Lights, Multiple Lights, and Ambient Light

Coloured Lights, Multiple Lights, and Ambient Light

- To get coloured lights, we perform lighting calculation three times to get an RGB triple.

- More correct to use spectra, and better approximations to spectra exist, but RGB sufficient for now
- To get multiple lights, compute contribution independently and sum:

$$L_{\text{out}}(\vec{v}) = \sum_i \rho(\vec{v}, \vec{\ell}_i) I_i \frac{\vec{\ell}_i \cdot \vec{n}}{c_1 + c_2 r_i + c_3 r_i^2}$$

- Question: what do pictures with this illumination look like?

Ambient Light:

- Lighting model so far is still too harsh
- Problem is that only direct illumination is modeled
- Global illumination techniques address this—but are expensive
- Ambient illumination is a simple approximation to global illumination
- Assume everything gets uniform illumination in addition to direct illumination

$$L_{\text{out}}(\vec{v}) = k_a I_a + \sum_i \rho(\vec{v}, \vec{\ell}_i) I_i \frac{\vec{\ell}_i \cdot \vec{n}}{c_1 + c_2 r_i + c_3 r_i^2}$$

16.5 Specular Reflection

Specular Reflection

- Lambertian term models matte surface but not shiny ones
- Shiny surfaces have "highlights" because energy reflected depends on viewer's position
- Phong Bui-Tuong developed an empirical model:

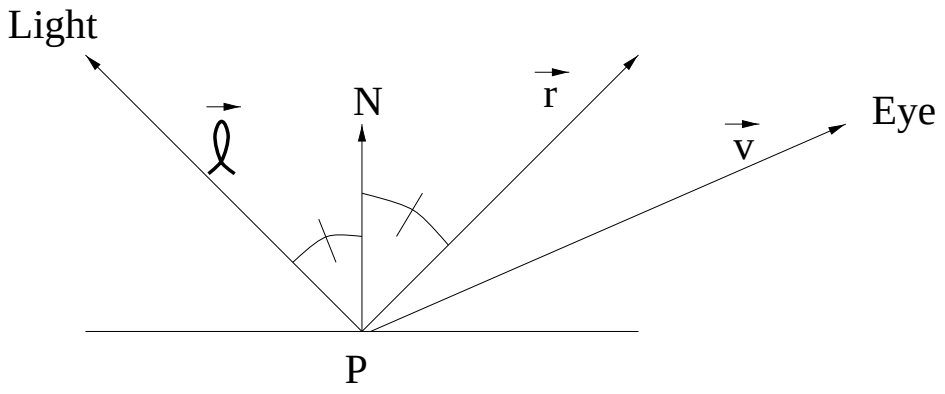
$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d (\vec{\ell} \cdot \vec{n}) I_d + k_s (\vec{r} \cdot \vec{v})^p I_s$$

- Using our previous notation:

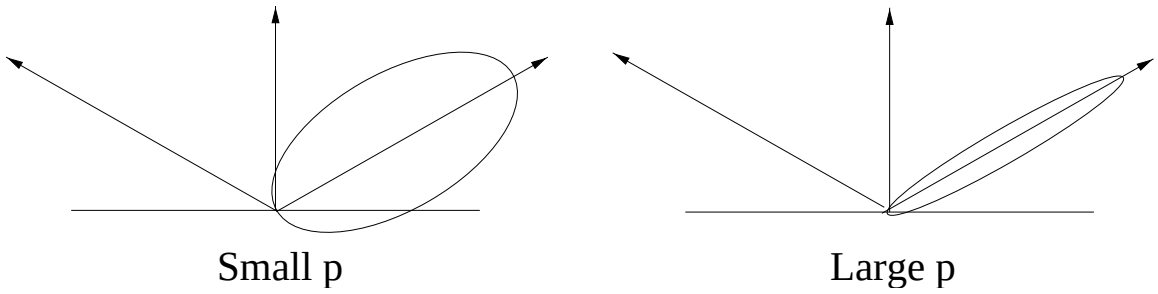
$$\rho(\vec{v}, \vec{\ell}) = k_d + k_s \frac{(\vec{r} \cdot \vec{v})^p}{\vec{n} \cdot \vec{\ell}}$$

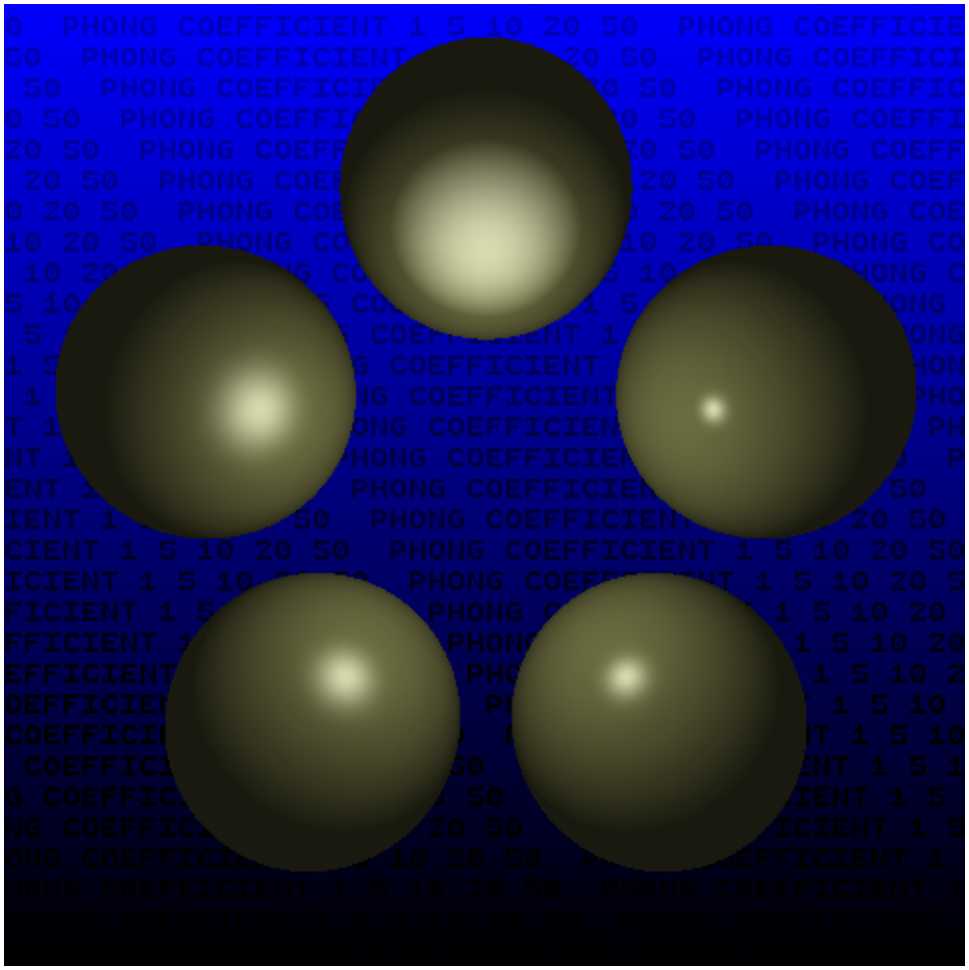
- The vector $\vec{r}$ is $\vec{\ell}$ reflected by the surface:

$$\vec{r} = -\vec{\ell} + 2(\vec{\ell} \cdot \vec{n})\vec{n}$$



- This is the classic *Phong lighting model*
- Specular term at a maximum (k_s) when $\vec{v} = \vec{r}$
- The exponent p controls sharpness of highlight:
  - Small p gives wide highlight
  - Large p gives narrow highlight





- Blinn introduced a variation, the *Blinn-Phong lighting model*:

$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d(\vec{\ell} \cdot \vec{n}) I_d + k_s(\vec{h} \cdot \vec{n})^p I_s$$

  Compare to Phong model of

$$L_{\text{out}}(\vec{v}) = k_a I_a + k_d(\vec{\ell} \cdot \vec{n}) I_d + k_s(\vec{r} \cdot \vec{v})^p I_s$$

- The halfway vector $\vec{h}$ is given by

$$\vec{h} = \frac{\vec{v} + \vec{\ell}}{|\vec{v} + \vec{\ell}|}$$

- Value $(\vec{h} \cdot \vec{n})$ measures deviation from ideal mirror configuration of $\vec{v}$ and $\vec{\ell}$
- Exponent works similarily to Phong model.
- OpenGL uses Blinn-Phong model.
- Blinn-Phong better motivated physically ...
- **BUT** both models break several laws of physics (conservation of energy, for instance).

- Better physically-based and empirical models exist:
  Cook-Torrance, He, Ward, Lafortune's modified Phong, etc.

- Good empirical model for project:
  Ashikhmin and Shirley,
  An Anisotropic Phong BRDF Model,
  Journal of Graphics Tools, AK Peters/ACM,
  Volume 5, Number 2, 2000.

17 Shading

17.1 Introduction

- Want to shade surfaces
- Lighting calculation for a point

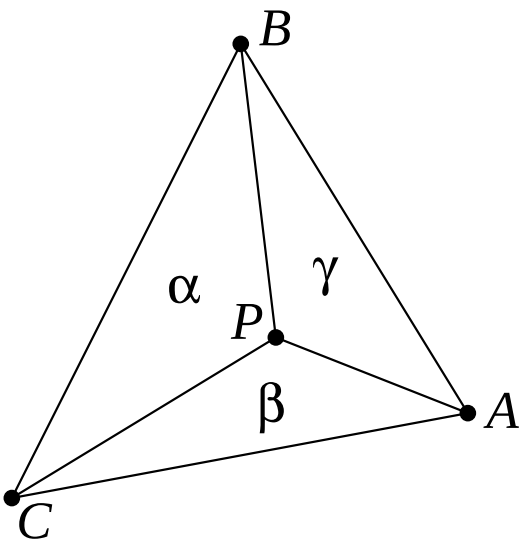
  Given: L_{in}, $\vec{\ell}$, and surface properties (including surface normal)
  Compute: L_{out} in direction $\vec{v}$
- Need surface normals at every point
- Commonly, surface is polygonal
  - True polygonal surface: use polygon normal
  - Sampled surface: sample position and normal, create polygonal approximation
- Want colour for each pixel in rasterized surface

Flat Shading

- Shade entire polygon one colour
- Perform lighting calculation at:
  - One polygon vertex
  - Center of polygon
    What normal do we use?
  - All polygon vertices and average colours
- Problem: Surface looks faceted
- OK if really is a polygonal model, not good if a sampled approximation to a curved surface.

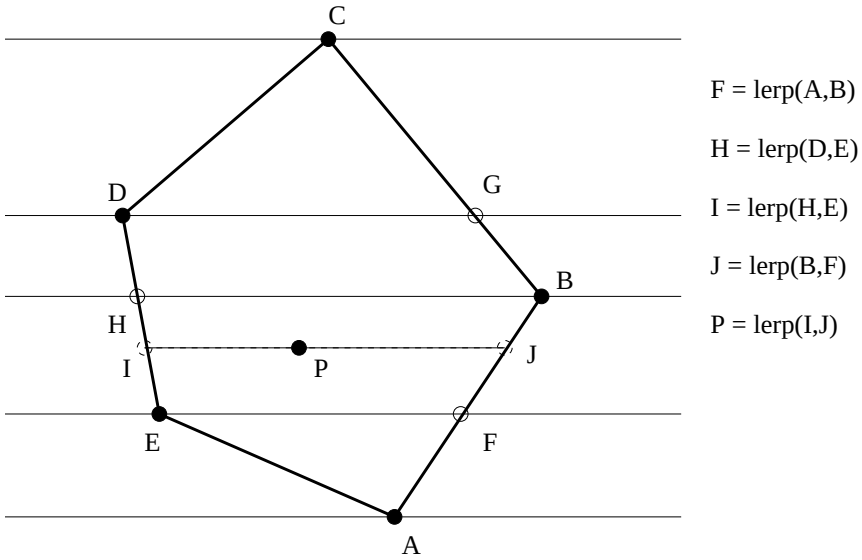
17.2 Gouraud Shading

- **Gouraud shading** interpolates colours across a polygon from the vertices.
- Lighting calculations are only performed at the vertices.
- Interpolation well-defined for triangles.
- Extensions to convex polygons . . . but not a good idea, convert to triangles.
- Barycentric combinations are also **affine combinations**. . .
  Triangular Gouraud shading is **invariant** under affine transformations.



$$\alpha = \triangle PBC / \triangle ABC$$
$$\beta = \triangle APC / \triangle ABC$$
$$\gamma = \triangle ABP / \triangle ABC$$
$$\alpha + \beta + \gamma = 1$$
$$P = \alpha A + \beta B + \gamma C$$

- To implement, can use repeated affine combination along edges, across spans, during rasterization.
- Gouraud shading is well-defined only for triangles
- For polygons with more than three vertices:
  - Sort the vertices by y coordinate.
  - Slice the polygon into trapezoids with parallel top and bottom.
  - Interpolate colours along each edge of the trapezoid...
  - Interpolate colours along each scanline.



- Gouraud shading gives **bilinear** interpolation within each trapezoid.

- Since rotating the polygon can result in a different trapezoidal decomposition, n-sided Gouraud interpolation is **not affine invariant.**
- Aliasing also a problem: highlights can be missed or blurred.

  Not good for shiny surfaces unless fine polygons are used.
- Exercise: Provide an example of the above effect.
- Exercise: Prove repeated affine combinations (the above algorithm) is equivalent to barycentric combinations on triangles.
- Exercise: Prove that the extension of the repeated affine combination algorithm to arbitrary polygons is not invariant under affine transformations.
- Common in hardware renderers; the model that (classic) OpenGL supports.
- Linear interpolation in device space not consistent with linear interpolation in world space. Modern implementations of OpenGL actually implement rational linear interpolation, which takes perspective into account.

17.3 Phong Shading

- **Phong Shading** interpolates lighting model parameters, **not** colours.
- Much better rendition of highlights.
- A **normal** is specified at each vertex of a polygon.
- Vertex normals are independent of the polygon normal.
- Vertex normals should relate to the surface being approximated by the polygonal mesh.
- The normal is interpolated across the polygon (using Gouraud techniques).
- At each pixel,
  - Interpolate the normal...
  - Interpolate other shading parameters...
  - Compute the view and light vectors...
  - Evaluate the lighting model.
- The lighting model does not have to be the Phong lighting model!
- Normal interpolation is nominally done by vector addition and renormalization.
- Several "fast" approximations are possible.
- The view and light vectors may also be interpolated or approximated.
- Problems with Phong shading:
  - Distances change under perspective transformation

  - Where do we do interpolation?
  - Normals don't map through perspective transformation
  - Can't perform lighting calculation or linear interpolation in device space
  - Have to perform lighting calculation in *world space* or *view space*, assuming model-view transformation is affine.
  - Have to perform linear interpolation in world or view space, project into device space
  - Results in rational-linear interpolation in device space!
  - Interpolate homogeneous coordinates, do per-pixel divide.
  - Can be organized so only need *one* division per pixel, regardless of the number of parameters to be interpolated.

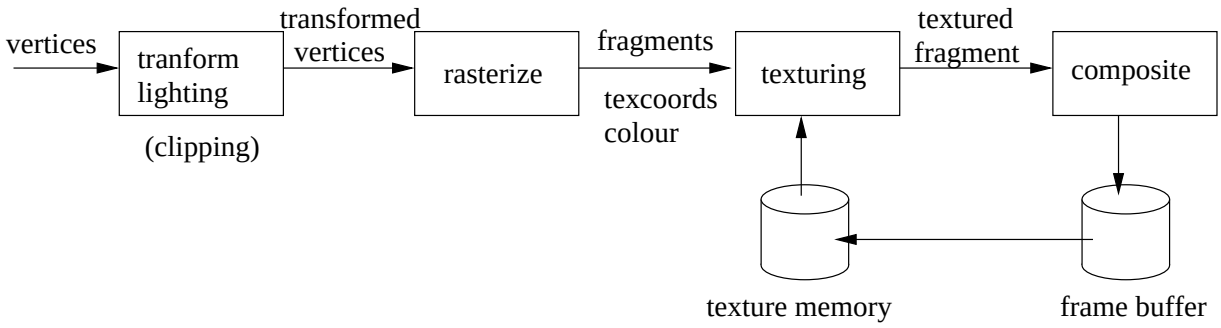
- Phong shading, and many other kinds of advanced shading, can be simulated with programmable vertex and fragment shaders on modern graphics hardware:
  - Classic Gouraud shading is linear in device space.
  - Modern graphics hardware performs rational linear interpolation — looks like interpolation happens in world space.
  - Interpolate normals, view vectors, and light vectors using generic interpolation hardware
  - Usually also interpolate diffuse term after computing it in vertex shader.
  - Write fragment shader to renormalize normals to unit length, compute lighting model, and assign colour to pixel.
  - Can also use texture maps as lookup tables...

18 Graphics Hardware

18.1 Graphics Hardware

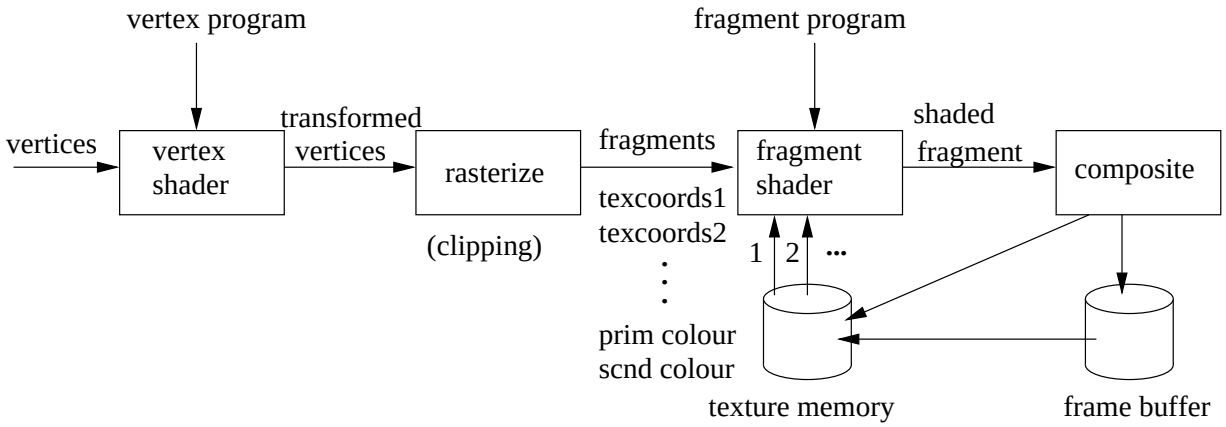
- Graphics hardware has been evolving over the past several decades
- In the 60's, 70's, vector displays costing millions of dollars
  US military was main (only) customer
- Raster displays started appearing in the 70's
- In the 80's, raster displays and graphics hardware cost 10's to 100's of thousands of dollars.
  Cost low enough that companies could afford the hardware.
  SGI won the battle of work station graphics, OpenGL became standard
- In the 90's, graphics hardware started moving to PC's
  DirectX is MicroSoft's challenge to OpenGL
- In the 00's, PC graphics hardware has displaced work station graphics
  Good, inexpensive cards for PCs
- Currently getting faster in excess of Moore's law
- Many graphics cards are "more powerful" than CPU in floating-point
- Can be used for general purpose computation too! PCI-express makes graphics accelerators like coprocessors

Traditional OpenGL (1.2)



- A fragment is a sort of virtual pixel
- fragment carries colour that is composited onto pixel in frame buffer if it passes depth, stencil, and alpha tests.
- Clipping done in transform/lighting portion

OpenGL (2.0)



- Vertex Shader and Fragment Shader are programmable by application

GeForce 3/4:

- 100's of lines of assembly for vertex shader (floating point 4-tuple)
- Configurable pipeline for fragment shader
  - 4 texture shader stages (floating point)
  - 8 register combiner stages (10-bit signed int)

ATI 9700

- 1000's of lines of assembly for both vertex and fragment shader
- Single precision float for vertex shader
- 24-bit float in fragment shader
  15-bit significand

GeForce 5x00:

- Single precision floating point in vertex shaders
- Half and single precision floating point in fragment shaders
- Floating point textures and rendering targets
- Unlimited number of dependent texture reads in fragment shader

ATI 8x0

- Single precision float for vertex shader
- 24-bit float in fragment shader
- Multiple floating point rendering targets supported

GeForce 6x00:

- Single precision floating point in vertex shaders
- Half and single precision floating point in fragment shaders
- Floating point textures and rendering targets
- Filtering and interpolation support for half-float textures
- Multiple rendering targets
- Control constructs (loops and conditionals) in both vertex and fragment programs
- Texture access in vertex unit (no filtering)

OpenGL 2.0

- Full single-precision floating point in both shaders
- Unlimited number of instructions
- Conditional execution and loops
- High level shading language in driver

Full support only available on 3Dlabs Realizm cards, so far...

18.2 NVIDIA Vertex Shader Assembly

Vertex Shader

- Sets of four word registers
- Data passed in through v, c registers
  Results passed to next stage through o registers
  - 16 vertex attribute registers
    v[0],...,v[15]
    Mnemonic names: v[OPOS], v[WGHT], V[NML],...
    position, weight, normal, primary color,...
    Programmer responsible for loading data into them
    Vertex attribute registers are read only
    Only one v register per instruction
- 15 vertex result registers
  o[HPOS] – Homogeneous clip space position
  o[COL0] – Primary colour
  o[TEX0] – Texture coordinate set 0
  ...
  Vertex result registers are write only
  Vertex program *must* write to o[HPOS]

- 12 temporary registers, R0,...,R11

  Read/write

- A0.x is the "address register"

  Write only

  Used to access constant registers

- 96 constant registers, c[0],...,c[95]

  May also address as c[A0.x]

  Read-only to normal program

  Read/write to vertex state program

  Only one constant register per instruction

- 17 assembly instructions

  - Operate on scalar(s) or 4-component register
  - Result is for single register

    Either 4-component or replicated scalar
  - Examples:

Opcode	Inputs	Output	Operations
MOV	v	v	move
ADD	v,v	v	add
RSQ	s	ssss	reciprocal square root
DP4	v,v	ssss	4-component dot product

  - Scalar input must have modifier .x, .y, .z, or .w
  - Swizzle: .xxyy, .wzyx, etc.

- Generate id, load/compile vertex program, etc.,...

  glGenProgramsNV, glLoadProgramNV,...

- Web pages:

  `http://developer.nvidia.com/docs/IO/1239/ATT/VertexPrograms.pdf`
  `http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program.txt`
  `http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program1_1.txt`

Example: Phong Lighting

```
// X = Vertex position
// LP = Light position
// EP = Eye position
// L = Normalized light vector
// E = Normalized eye vector
// R = Reflection vector
// n = Specular exponent

// Parameters:
//   c[0-3] = Modelview times projection matrices
//   c[10] = Diffuse color
```

```
//   c[11] = Specular color
//   c[12] = Light position
//   c[13] = Eye position
//   c[14] = (n, 0.0, 0.0, 0.0)
//   c[16] = (2.0, 2.0, 2.0, 2.0)

GLubyte vpPhong_str[] =
"!!VP1.0\n"

"DP4 o[HPOS].x, c[0], v[OPOS];"  // Apply modelview and
"DP4 o[HPOS].y, c[1], v[OPOS];"  // projection matrices
"DP4 o[HPOS].z, c[2], v[OPOS];"
"DP4 o[HPOS].w, c[3], v[OPOS];"

"ADD R0, c[12], -v[OPOS];"    // R0 = LP - X
"DP3 R1, R0, R0;"             // R1 = |LP - X|^2
"RSQ R2, R1.w;"               // R2 = 1/|LP - X|
"MUL R0, R0, R2;"             // R0 = L

"DP3 R3, R0, v[NRML];"        // R3 = N.L

"ADD R4, c[13], -v[OPOS];"    // R4 = EP - X
"DP3 R5, R4, R4;"             // R5 = |EP - X|^2
"RSQ R6, R5.w;"               // R6 = 1/|EP - X|
"MUL R4, R4, R6;"             // R4 = E

"DP3 R7, R4, v[NRML];"        // R7 = E.N
"MUL R7, R7, c[16];"          // R7 = 2*(E.N)
"MAD R8, R7, v[NRML], -R4;"   // R8 = 2*(E.N)N-E = R

"DP3 R9, R8, R0;"             // R9 = R.L

"LOG R10, R9.x;"              // R10.z = LOG(R.L)
"MUL R9, c[14].x, R10.z;"     // R9 = n*LOG(R.L)
"EXP R11, R9.z;"              // R11 = EXP(n*LOG(R.L)) = (R.L)^n

"MUL R10, R3, c[10];"               // R10 = Cd*(N.L)
"MAD o[COL0], c[11], R11.z, R10;"   // o[COL0] = Cd*(N.L) + Cs*(R.L)^n

"END";
```

18.3 High-Level Shading Languages

- OpenGL, DirectX APIs support assembly interface for vertex and fragment programs
- Hard to program (maintain, read) assembly
- Several high level shading languages now available: Cg, GLSL, Microsoft HLSL, Sh

18.4 Cg

- Cg is an example of a C-like language to create shaders

- Created and supported by NVIDIA, but can target other cards
- Also API independent: OpenGL or DirectX
- Compile in advance (to assembly) or at runtime
- Web page for docs http://developer.nvidia.com/Cg

Cg example (from nvidia Cg Toolkit User's Manual)
Have to define inputs and outputs:

```
// define inputs from application
struct appin {
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};

// define outputs from vertex shader
struct vertout {
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
};

vertout main(appin In,
         uniform float4x4 ModelViewProj : C0,
         uniform float4x4 ModelViewIT : C4,
         uniform float4 LightVec)
{
    vertout Out;
    Out.Hposition = mul(ModelViewProj, In.Position);
    // transform normal from model space to view space
    float4 normal = normalize(mul(ModelViewIT, In.Normal).xyzz);

    // store normalized light vector
    float4 light = normalize(LightVec);

    // calculate half angle vector
    float4 eye = float4(0.0, 0.0, 1.0, 1.0);
    float4 half = normalize(light + eye);

    // calculate diffuse component
    float diffuse = dot(normal, light);

    // calculate specular component
    float specular = dot(normal, half);
    specular = pow(specular, 32);

    // blue diffuse material
    float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);

    // white specular material
    float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);
```

```
    // combine diffuse and specular contributions
    // and output final vertex color
    Out.Color0 = diffuse * diffuseMaterial + specular * specularMaterial;
    return Out;
}
```

18.5 Sh

- Sh embeds the shading language in a C++ API
- Developed at the University of Waterloo
- API independent: OpenGL or DirectX
- Compiles at runtime
- Can also generate code for host CPU
- Can also be used for general-purpose computation
- Web page: http://libsh.org

Sh example (following Cg example)

Have to define global parameters (a little more general than Cg shader, have converted hard-coded values to parameters):

```
// transformation matrices
ShMatrix4x4f ModelViewProj;
ShMatrix4x4f ModelViewIT;

// light direction vector
ShVector3f LightVec;

// blue diffuse material
ShColor4f diffuseMaterial(0.0, 0.0, 1.0, 1.0);

// white specular material
ShColor4f specularMaterial(1.0, 1.0, 1.0, 1.0);

// specular power
ShAttrib1f spec_power = 32;

ShProgram vs = SH_BEGIN_PROGRAM("gpu:vertex") {
    ShInputPosition4f position;
    ShInputNormal3f normal;

    ShOutputPosition4f Hposition;
    ShOutputColor4f Color0;

    Hposition = ModelViewProj | position;
```

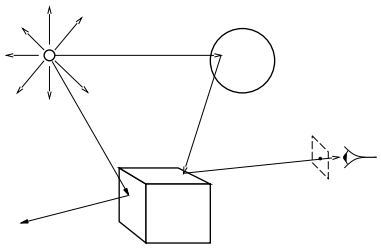
```
    // transform normal from model space to view space
    normal = normalize(ModelViewIT | normal)(0,1,2));

    // store normalized light vector (will be lifted to host)
    ShVector3f light = normalize(LightVec);

    // calculate half angle vector
    ShVector3f eye(0.0, 0.0, 1.0);
    ShVector3f half = normalize(light + eye);

    // calculate diffuse component
    ShAttrib1f diffuse = (normal | light);

    // calculate specular component
    ShAttrib1f specular = (normal | half);
    specular = pow(specular, spec_power);

    // combine diffuse and specular contributions
    // and output final vertex color
    Color0 = diffuse * diffuseMaterial + specular * specularMaterial;
} SH_END;
// load it into vertex shader
shBind(vs);
```

19 Ray Tracing

19.1 Fundamentals

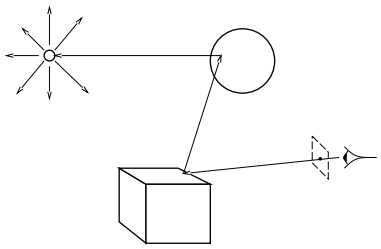
Ray Tracing: Photorealism – The Holy Grail

- Want to make more realistic images
  - Shadows
  - Reflections
- What happens in the real world?



Problem: Many rays never reach the eye.

- Idea: Trace rays backwards



Ray Tracing: Basic Code

```
foreach pixel
    ray = (eye, pixel-eye);
    Intersect(Scene, ray);
end
```

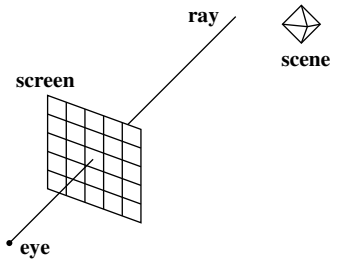
Issues:

- Which pixels?
- How do we perform intersections?
- How do we perform shading?
- How can we speed things up?
- How can we make things look better?

Ray Tracing/Casting: Selecting the initial ray

- Setting: *eyepoint*, *virtual screen* (an array of *virtual pixels*, and *scene* are organized in convenient coordinate frames (e.g. all in view or world frames)

- Ray: a half line determined by the eyepoint and a point associated with a chosen pixel
- Interpretations:
  - Ray is the path of photons that succesfully reach the eye
    (we simulate selected photon transport through the scene)
  - Ray is a sampling probe that gathers color/visibility information



(Readings: Watt: 12.1. Red book: 14.7. White book: 16.12.)

19.2 Intersection Computations

Intersection Computations

General Issues:

- Ray: express in parametric form

$$E + t(P - E)$$

  where E is the eyepoint and P is the pixel point
- Scene Object: direct implicit form
  - express as $f(Q) = 0$ when Q is a surface point, where f is a given formula
  - intersection computation is an equation to solve:
    find t such that $f(E + t(P - E)) = 0$

- Scene Object: procedural implicit form
  - f is not a given formula
  - f is only defined procedurally
  - $\mathcal{A}(f(E + t(P - E)) = 0)$ yields t, where A is a root finding method
    (secant, Newton, bisection, ...)

Quadric Surfaces

Surface given by

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + Gx + Hy + Jz + K = 0$$

- Ellipsoid: $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$

Ellipsoid

- Elliptic paraboloid: $\frac{x^2}{p^2} + \frac{y^2}{q^2} = 1$

Elliptic paraboloid

- Hyperboloid of one sheet: $\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 1$

Hyperboloid of one sheet

- Cone: $\frac{x^2}{p^2} + \frac{y^2}{q^2} - \frac{z^2}{r^2} = 0$

Cone

- Elliptic cylinder: $\frac{x^2}{p^2} + \frac{y^2}{q^2} = 1$

Elliptic cylinder

- Ray given by

$$\begin{aligned} x &= x_E + t(x_P - x_E) \\ y &= y_E + t(y_P - y_E) \\ z &= z_E + t(z_P - z_E) \end{aligned}$$

- Substitute ray x,y,z into surface formula
  - quadratic equation results for t
  - organize expression terms for numerical accuracy; ie. to avoid
    - cancellation
    - combinations of numbers with widely different magnitudes

Polygons:

- The plane of the polygon should be known

$$Ax + By + Cz + D = 0$$

  - $(A, B, C, 0)$ is the normal vector

  - pick three succesive vertices

$$\begin{aligned} v_{i-1} &= (x_{i-1}, y_{i-1}, z_{i-1}) \\ v_i &= (x_i, y_i, z_i) \\ v_{i+1} &= (x_{i+1}, y_{i+1}, z_{i+1}) \end{aligned}$$

  - should subtend a "reasonable" angle (bounded away from 0 or 180 degrees)
  - normal vector is the cross product $(v_{i_1} - v_i) \times (v_{i-1} - v_i)$
  - $D = -(Ax + By + Cz)$ for any vertex $(x, y, z, 1)$ of the polygon

- Substitute ray x,y,z into surface formula
  - linear equation results for t
- Solution provides planar point $(\bar{x}, \bar{y}, \bar{z})$
  - is this inside or outside the polygon?

Planar Coordinates:

- Take origin point and two independent vectors on the plane

$$\begin{aligned} \mathcal{O} &= (x_{\mathcal{O}}, y_{\mathcal{O}}, z_{\mathcal{O}}, 1) \\ \vec{b_0} &= (u_0, v_0, w_0, 0) \\ \vec{b_1} &= (u_1, v_1, w_1, 0) \end{aligned}$$

- Express any point in the plane as

$$P - \mathcal{O} = \alpha_0 \vec{b_0} + \alpha_1 \vec{b_1}$$

  - intersection point is $(\bar{\alpha}_0, \bar{\alpha}_1, 1)$
  - clipping algorithm against polygon edges in these α coordinates

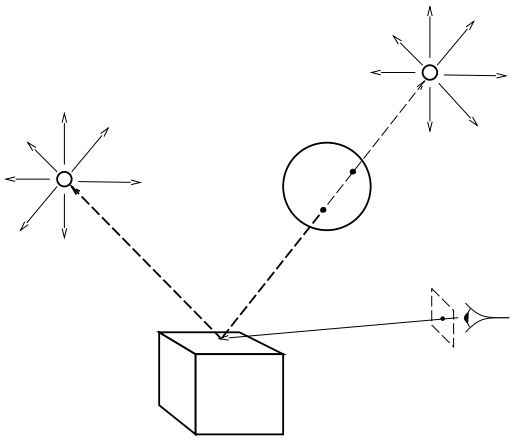
Alternatives: Must this all be done in world coordinates?

- Alternative: Intersections in model space
  - form normals and planar coordinates in model space and store with model
  - backtransform the ray into model space using inverse modeling transformations
  - perform the intersection and illumination calculations
- Alternative: Intersections in world space
  - form normals and planar coordinates in model space and store
  - forward transform using modeling transformations

19.3 Shading

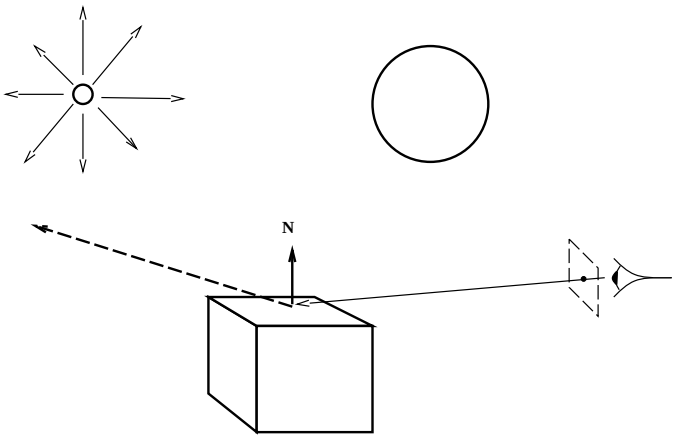
Shading, shadows

- How do we use ray to determine shade of pixel?
- At closest intersection point, perform Phong shading.
  Gives same images as polygon renderer, but more primitives.
- Idea: Before adding contribution from a light, cast ray to light.
  - If ray hits object before light, then don't shade.
  - This gives shadows.

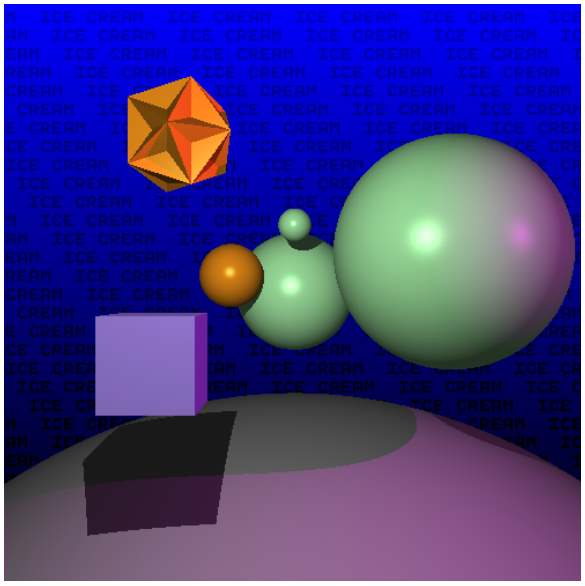
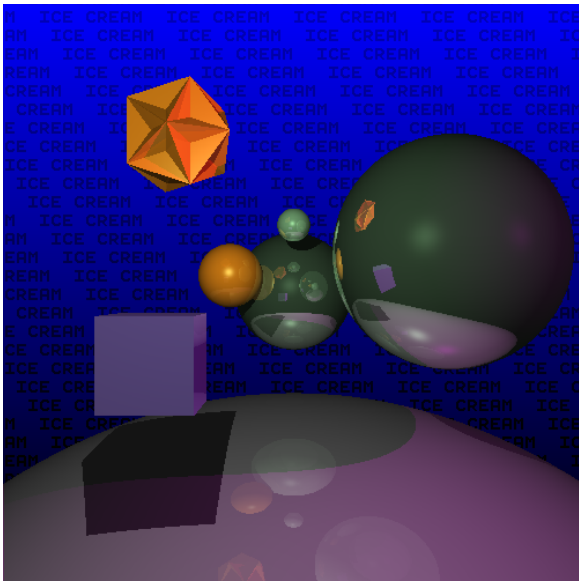


Reflections

- Idea: Cast a ray in the mirror direction.
- Add the colour coming from this ray to the shade of the initial ray.
- This gives mirror reflections



- We perform a full lighting calculation at the first intersection of this reflected ray.

(Readings: Watt: 12.1.)

19.4 Recursive Ray Tracing

Ray Tracing: Recursive

- Eye-screen ray is the *primary* ray
- Backward tracking of photons that could have arrived along primary
- Intersect with all objects in scene
- Determine nearest object
- Generate secondary rays
  - to light sources
  - in reflection-associated directions
  - in refraction-associated directions
- Continue recursively for each secondary ray
- Terminate after suitably many levels
- Accumulate suitably averaged information for primary ray
- Deposit information in pixel

Ray Casting: Non-recursive

- As above for ray tracing, but stop before generating secondary rays
- Apply illumination model at nearest object intersection with no regard to light occlusion
- Ray becomes a sampling probe that just gathers information on
  - visibility
  - color

(Readings: Watt: 12.2.)

19.5 Surface Information

Surface Normals:

- Illumination models require:
  - surface normal vectors at intersection points
  - ray-surface intersection computation must also yield a normal
  - light-source directions must be established at intersection
  - shadow information determined by light-ray intersections with other objects
- Normals to polygons:
  - provided by planar normal
  - provided by cross product of adjacent edges
  - Or use Phong normal interpolation if normals specified at vertices

- Normals to any implicit surface (eg. quadrics)
  - move from (x, y, z) to $(x + \Delta x, y + \Delta y, z + \Delta z)$ which is *maximally* far from the surface
  - direction of greatest increase to $f(x, y, z)$
- Taylor series:

$$f(x + \Delta x, y + \Delta y, z + \Delta z) = f(x, y, z) + [\Delta x, \Delta y, \Delta z] \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} + \cdots$$

  - maximality for the *gradient vector* of f
  - not normalized
- Normal to a quadric surface

$$\begin{bmatrix} 2Ax + By + Cz + G \\ Bx + 2Dy + Ez + H \\ Cx + Ey + 2Fz + J \\ 0 \end{bmatrix}$$

Normal Transformations: How do affine transformations affect surface normals?

- Let P_a and P_b be any two points on object
  - arbitrarily close
  - $\vec{n} \cdot (P_b - P_a) = 0$
  - using transpose notation: $(\vec{n})^T(P_b - P_a) = 0$
- After an affine transformation on P_a,P_b:

$$\mathbf{M}(P_b - P_a) = \mathbf{M}P_b - \mathbf{M}P_a$$

  we want $(\mathbf{N}\vec{n})$ to be a normal for some transformation $\mathbf{N}$:

$$\begin{aligned} & (\mathbf{N}\vec{n})^T\mathbf{M}(P_b - P_a) = 0 \\ \Longrightarrow \quad & \vec{n}^T\mathbf{N}^T\mathbf{M}(P_b - P_a) \end{aligned}$$

  and this certainly holds if $\mathbf{N} = (\mathbf{M}^{-1})^T$

  - Only the upper 3-by-3 portion of $\mathbf{M}$ is pertinent for vectors

- Translation:

$$(\mathbf{M}^{-1})^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\Delta x & -\Delta y & -\Delta z & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$$

  $\Longrightarrow$ no change to the normal

- Rotation (example):

$$\begin{aligned} (\mathbf{M}^{-1})^T &= \begin{bmatrix} \cos(-\theta) & \sin(-\theta) & 0 & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^T \\ &= \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &\longrightarrow \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

  $\Longrightarrow$ rotation applied unchanged to normal

- Scale:

$$(\mathbf{M}^{-1})^T = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & \frac{1}{s_z} \end{bmatrix}$$

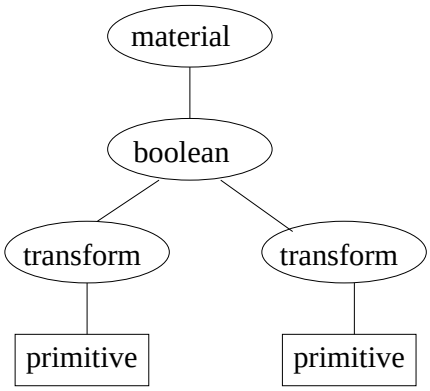
$$\begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & \frac{1}{s_z} \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{s_x} \\ \frac{n_y}{s_y} \\ \frac{n_z}{s_z} \\ 0 \end{bmatrix}$$

  $\Longrightarrow$ reciprocal scale applied to normal

19.6 Modeling and CSG

Modeling: Constructive Solid Geometry

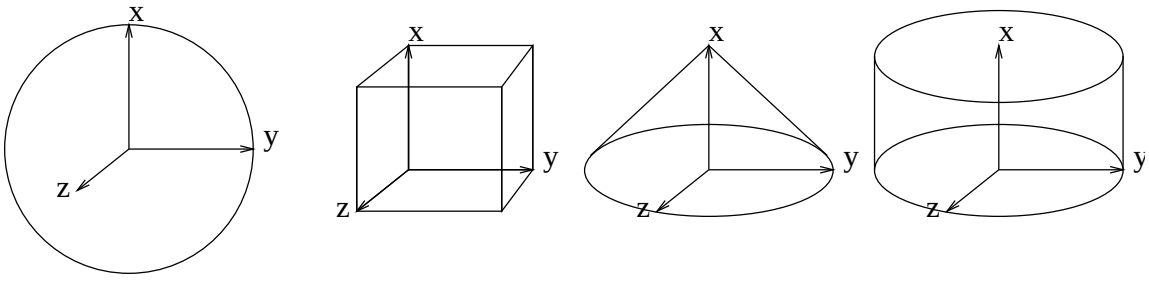
- How do we model for a Ray Tracer?
- Hierarchical modeling works well, but we can do more.
- In *Constructive Solid Geometry* all primitives are solids.
- New type of internal node: Boolean operation.
  Intersection, Union, Difference
- Thus, our model is a DAG with
  - Leaf nodes representing primitives
  - Interal nodes are transformations, materials, or boolean operations.



CSG: The primitives

We will want a rich set of primitives. How do we specify them?

- As a "canonical" primitive

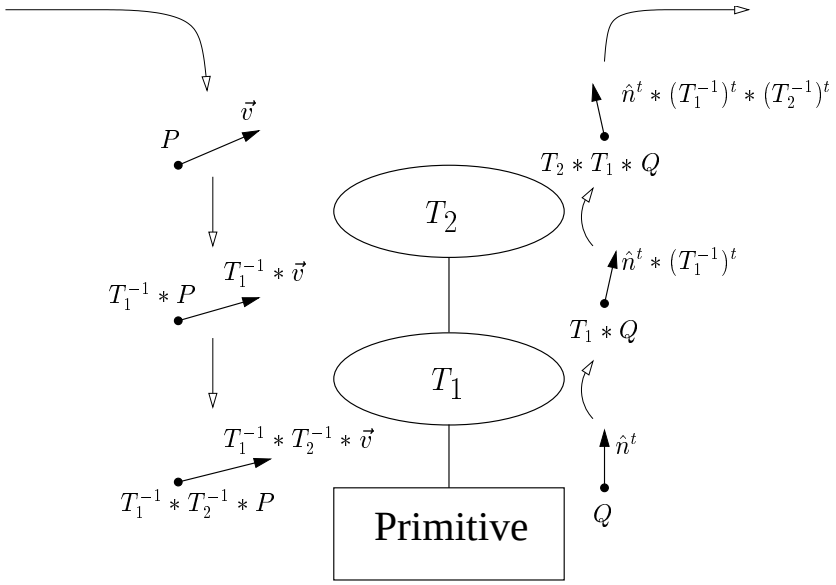


  Sphere()*Scale(2,2,2)*Translate(x,y,z)
- As a transformed canonical primitive
  - Sphere(r,x,y,z);
  - Cube(x,y,z,dx,dy,dz);
  - Cone(width,height,x,y,z,dx,dy,dz);

CSG: Traversing the transformation tree

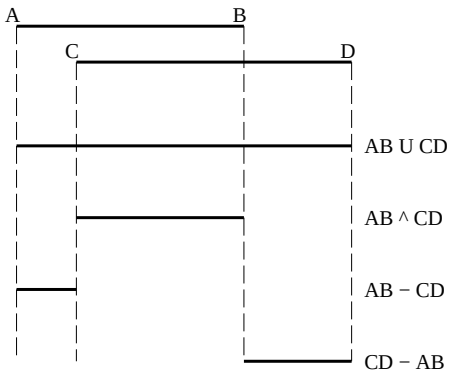
- After transformation applied, primitives will be warped.

- Rather than intersect ray with warped primitive, we transform the ray.
- On the way down, apply *inverse transformation* to ray.
- On the way back, apply transformation to the point, normal
  Caution: see notes on transforming normals

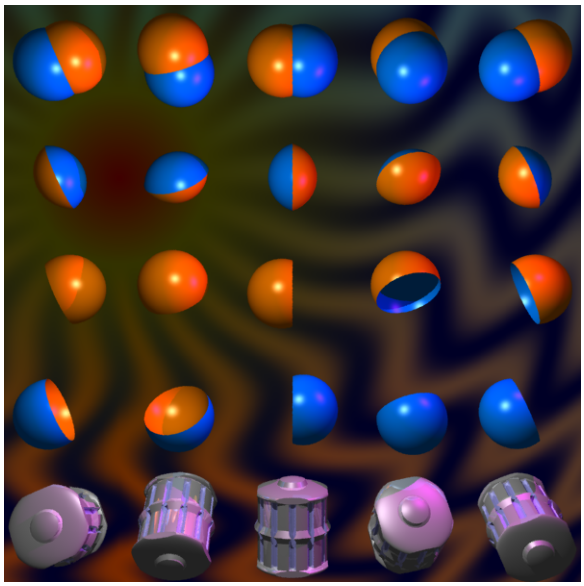


CSG: Boolean operations

- Could apply boolean operations to transformed primitives
  Problem: representation too complex
- Idea: perform a complete ray intersect object with each primitive. This gives us a set of line segment(s).
  Next, perform boolean operations on line segments.



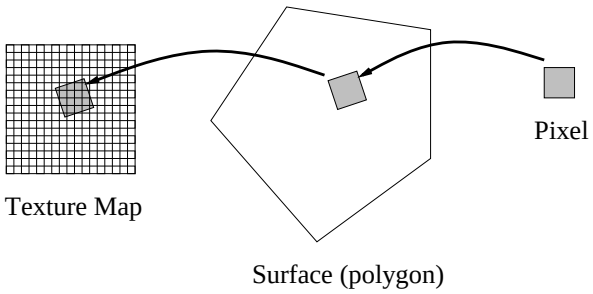
- Note that if we transform the ray on the way down, then we must transform the entire segment on the way back
- Also, we must be careful with normals: they flip for difference.



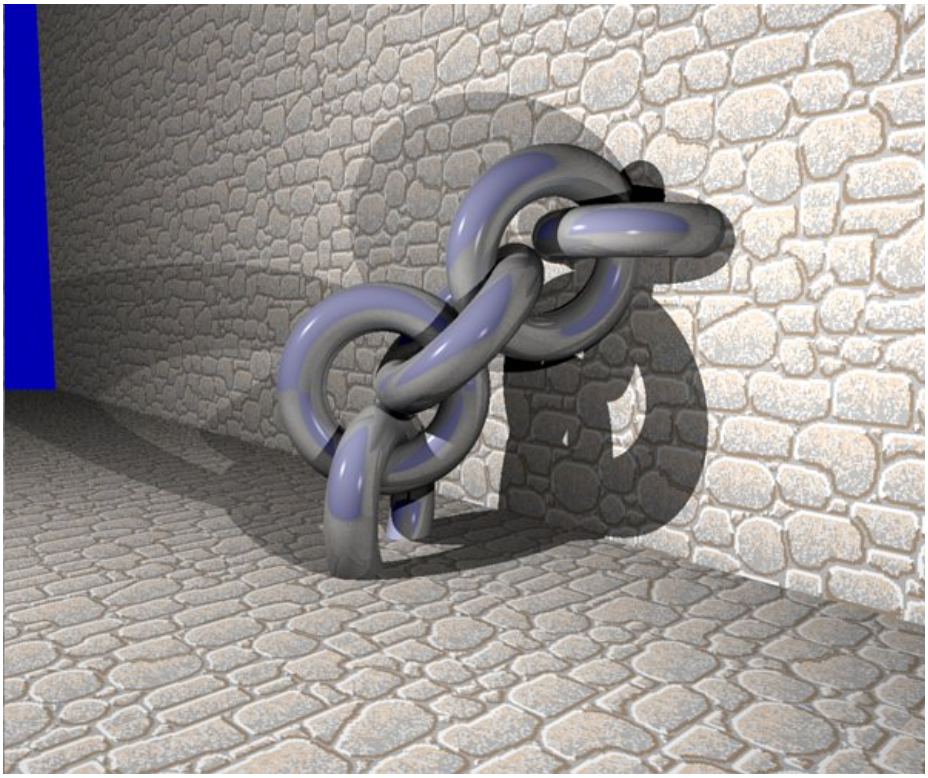
Examples: Union, Intersection, Difference, Difference, Complex Model

19.7 Texture Mapping

- If we add detail by increasing model complexity, then computation cost increases.
- If detail is surface detail, then we can use texture mapping.
- Idea: scan a photo of the detail and paste it on objects.
  - Associate texture with polygon
  - Map pixel onto polygon and then into texture map
  - Use weighted average of covered texture to compute colour.



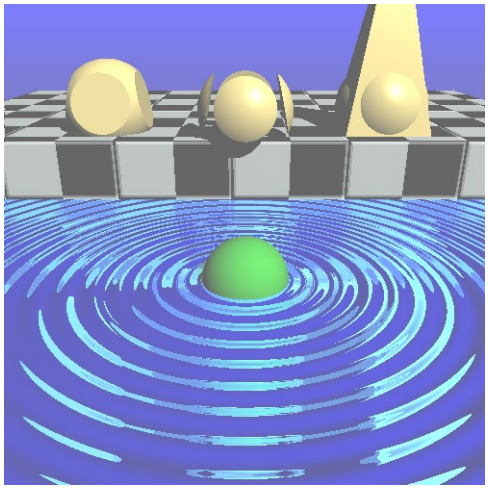
- Tile polygon with texture if needed
  - For some textures, tiling introduces unwanted pattern
  - Various method to extend textures
- Greatly improves images!
- Not just ray traced image



Bump Mapping

- Textures will still appear "smooth" (i.e., no shadows)
- Bump mapping is similiar to texture mapping except that we peturb the normal rather than the colour.

- Perturbed normal is used for lighting calculation.
- Convincing – usually fail to notice silhouette is wrong.



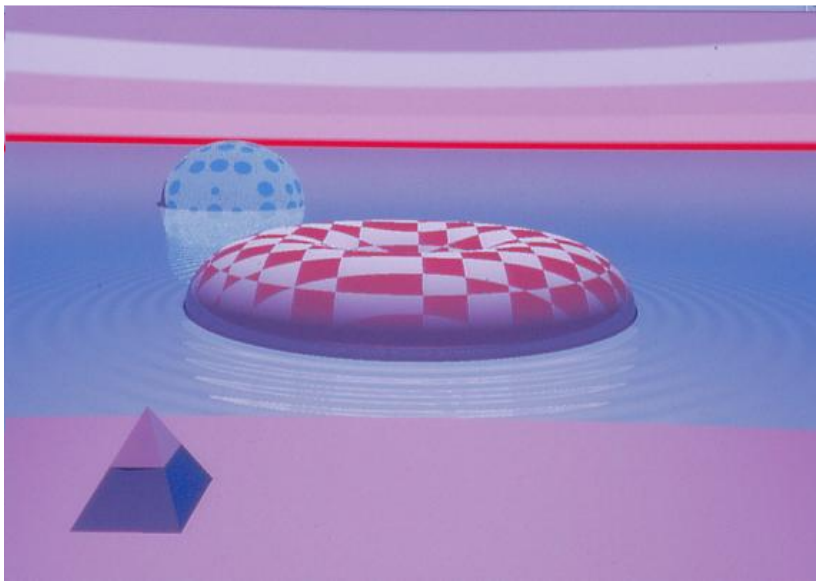
Solid Textures

- 2D textures can betray 2D nature in images
- Hard to texture map onto curved surfaces
- Idea: Use a 3D texture instead
- Usually procedural

  Example:

  ```
  if ( (floor(x)+floor(y)+floor(z))%2 == 0 ) then
      return RED;
  else
      return SILVER:
  end
  ```

  Gives a "3D checker board"
- Turbulence can also be simulated (marble)



(Readings: Watt: 8.)

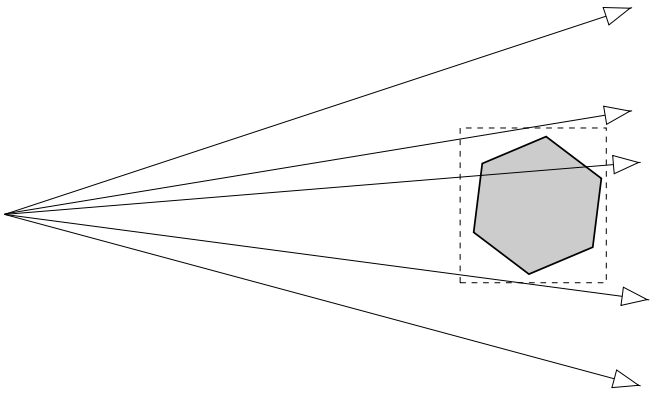
19.8 Bounding Boxes, Spatial Subdivision

Speed-ups

- Ray tracing is slow
- Ray intersect object is often expensive
- Improve speed two ways
  - Reduce the cost of ray intersect object
  - Intersect ray with fewer objects

Bounding Boxes

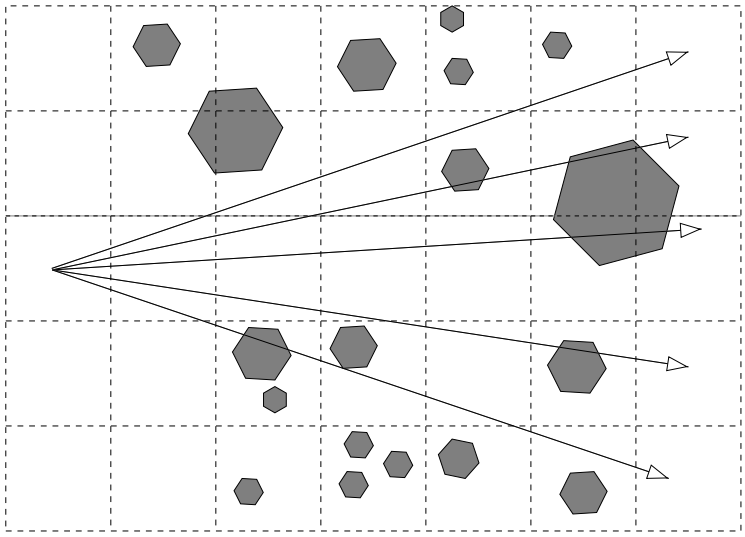
- Idea: Place a bounding box around each object
- Only compute ray intersect object if the ray intersects the bounding box



- If box aligned with coordinate axes, ray intersect box is very cheap
- Be careful with CSG/Hierarchical
- Can also use bounding spheres
- Most useful when ray intersect object is VERY expensive
  Example: polygonal object with lots of facets
- Construction of good bounding box can be difficult

Spatial Subdivision

- Idea: Divide space in to subregions
- Place objects from scene in to appropriate subregions
- When tracing ray, only intersect with objects in sub-regions through which the ray passes

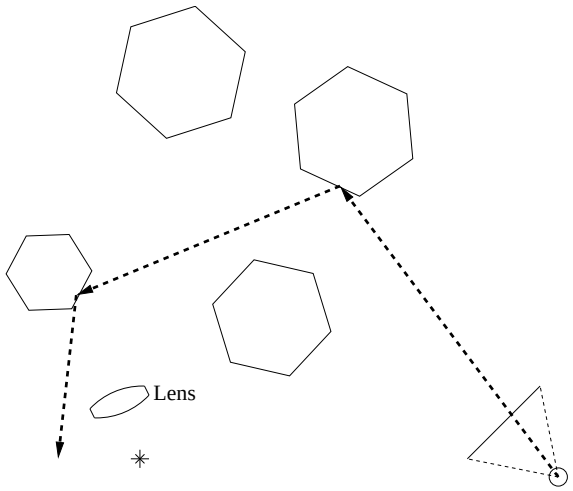


- Useful when lots of small objects in scene
- Determining subdivision is difficult

20 Bidirectional Tracing

20.1 Missing Effects

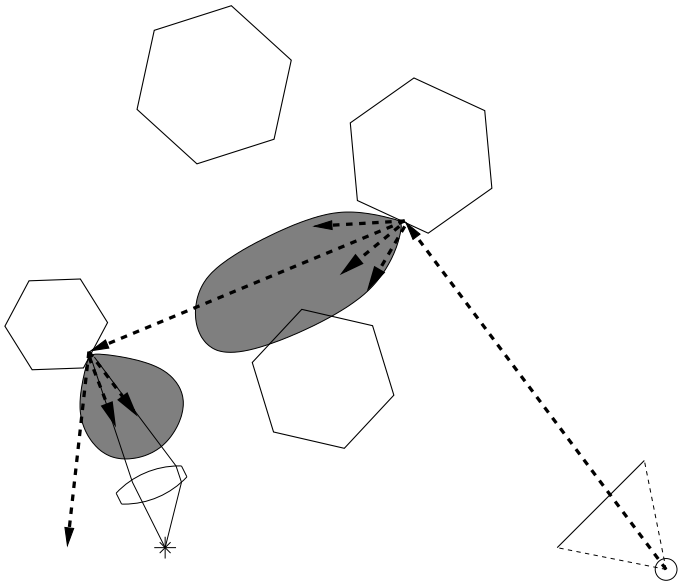
- Problem: Raytracing and radiosity can't model all effects
  Caustics



- Some effects requiring tracing light rays from light back to eye
- Too many to trace them all. How do we select?

20.2 Distribution Ray Tracing

- Ray trace, but at each reflection cast a lot of rays
- Cast with distribution function to represent reflection properties

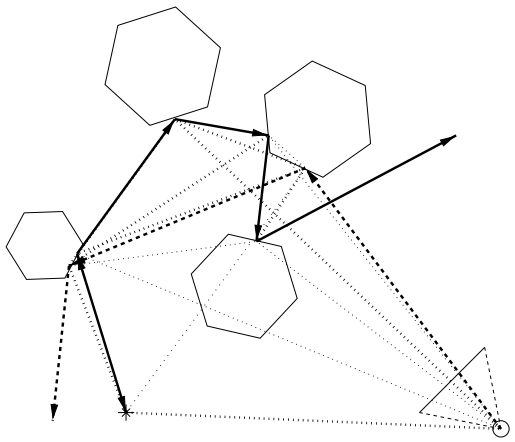


- Distributing rays over reflected direction gives soft reflections
  (similar effect if distributed over transmission direction)
- Distributing rays over light area gives soft shadows
- Distributing rays over aperature gives depth of field
- Distributing rays over time gives motion blur
- Can use to model caustics, but need *lots* of rays
  (20+ rays per reflection)
- Can model many effects simply, but often very expensive

Example: Distributed reflection
Example: Distributed aperture

20.3 Bidirectional Path Tracing

- Idea: trace paths from eye and from light and connect them up
  Trace paths (single reflected ray)
- Reflect with distribution function
- When far enough, connect them up
- Problem: A whole lot of rays

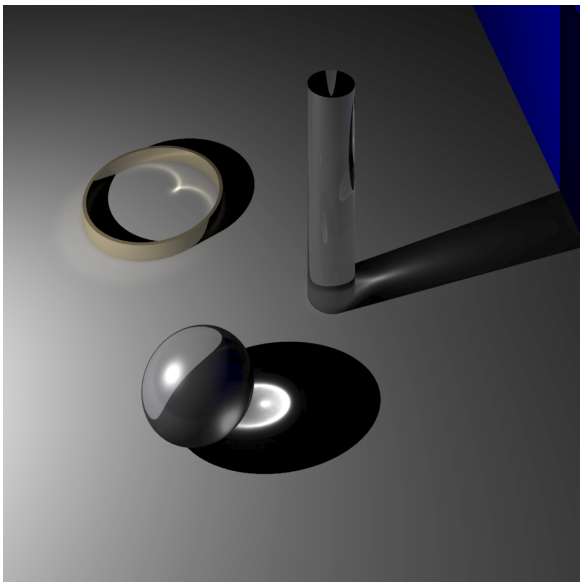


- Low error, high noise
- Less expensive than distribution ray tracing

20.4 Photon Maps

- Cast rays from light
  Global illumination (view independent)
- Create a "photon map" wherever this light hits
- The use standard ray *casting*,
  no secondary rays
  look at photon map(s) to determine special lighting
- Use density estimation to convert photon map to intensity
- Fast, easy to program, low noise
- Errors

Example: caustics

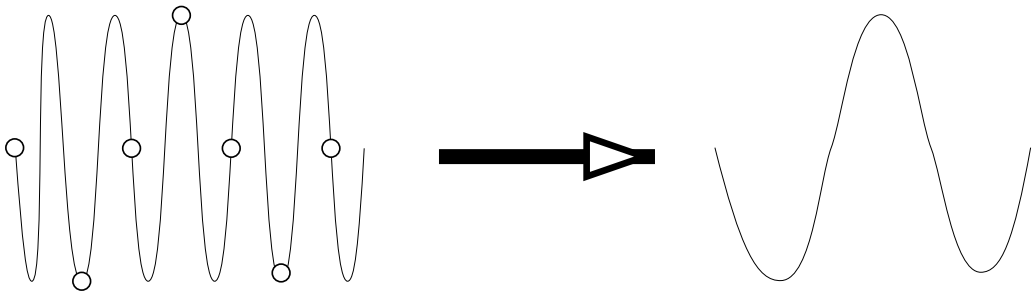


21 Aliasing

21.1 Aliasing

Aliasing and Anti-Aliasing

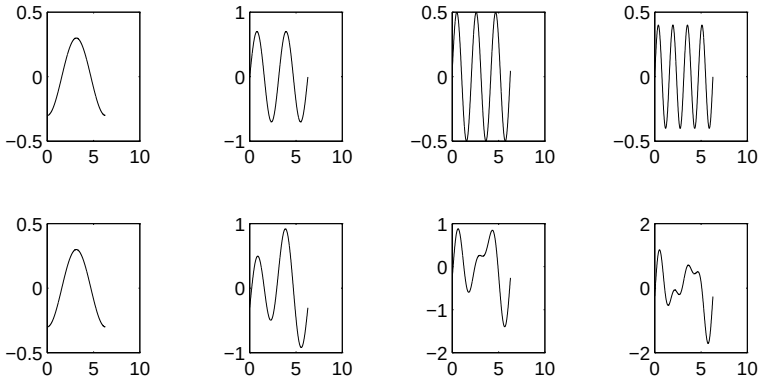
- Raster Image is a sampling of a continuous function
- If samples spaced too far apart, then don't get true representation of scene:



- In graphics, a variety of bad things happen:
  - Stairstep or "jaggies"
  - Moire Patterns
  - Loss of small objects
  - Temporal
    - Sampled too far apart in time
      Backwards rotating wheels
    - Crawling jaggies
    - Appearing/dissappearing objects, flashing objects

Image as a signal

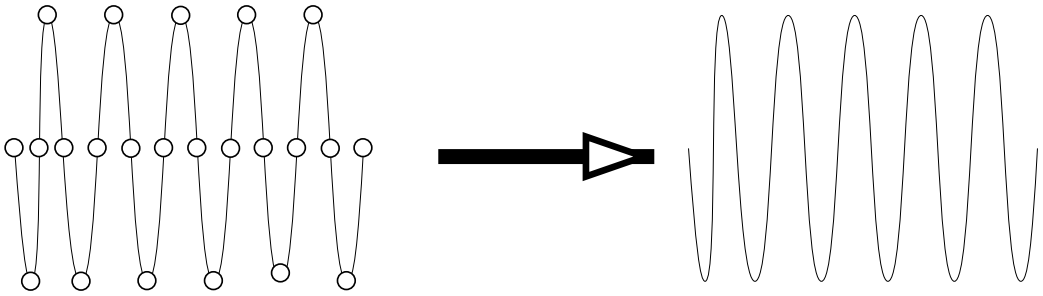
- Signal processing signal is function of time
- Scene is a function of space
  Spatial domain: $f(u)$
- Raster image is a sampling of this signal
- Can represent signal as sum of sine waves
  Frequency domain: $F(u)$



- Regular sampling restricts frequencies at which we sample

Nyquist Limit

- Theory tells us that we must sample at twice the highest frequency in the image to avoid aliasing.
- This sampling rate is known as the *Nyquist Limit*



- Problem: Man made objects have distinct edges

  Distinct edges have infinite frequency

What to do?

- After we get image, run through low pass filter

  This helps, but not a full solution
- Get a Higher Resolution Monitor

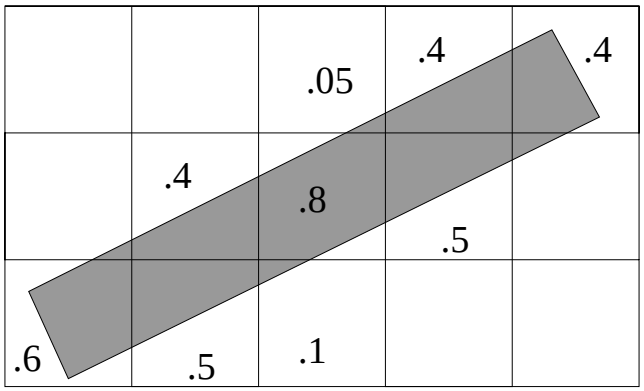
  This helps, but...
  - Not usually feasible
  - Alleviates jaggies but
    - Moire patterns merely shifted

    - Small objects still vanish
  - Increased CPU spent on extra pixels can be put to better use.
- Smarter sampling

Area Sampling

Rather than sample, we could integrate.

For lines, this means treating them as boxes



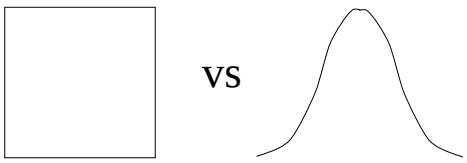
- Colour shades of gray based on fraction of pixel covered
- Gives better looking images *at a sufficiently far distance*
  Look close and it looks blurry

Weighted Sampling

- Unweighted sampling is a box filter

$$I = \int_{x \in \text{Box}} I_x dI$$

  - No contribution outside of pixel
  - All contributions are equal
- Weighted sampling
  - Give different weights depending on position in pixel



  - Filter may extend outside of pixel
    Avoids certain temporal aliasing problems
    "Correct" filter is infinite

Anti-aliasing in Ray Tracing

- Can't always integrate area under pixel
- For ray tracing, we want to point sample the image
- **Super Sampling**

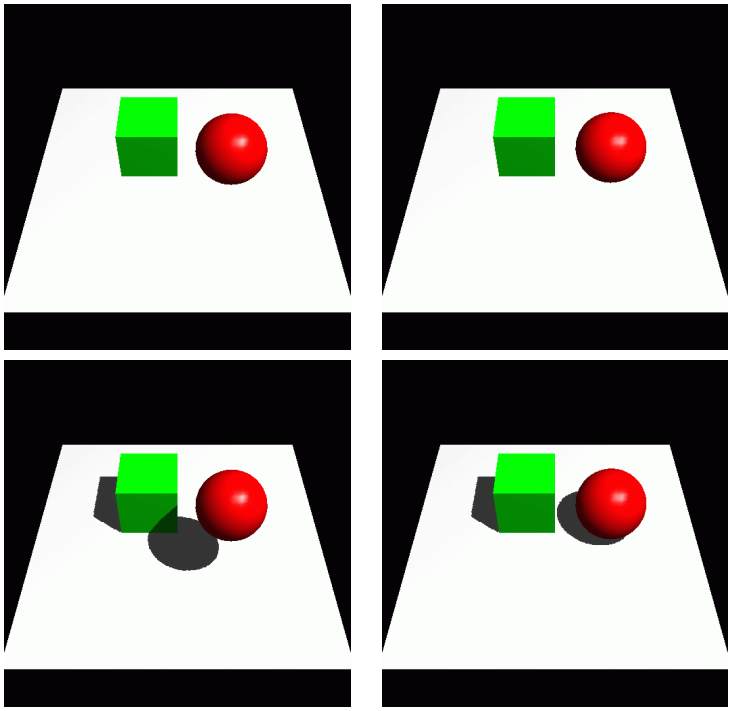
  Take more samples and weight with filter

  If sampling pattern regular, we still get aliasing
- **Stochastic Sampling**
  - Idea: Eye easily detects coherent errors (aliasing)
    Eye is poor at detecting incoherent errors (noise)
  - Rather than regular supersampling, we "jitter" the samples in one of several ways:
    * Choose random location within pixel
    * Displace small, random distances from regular grid

22 Shadows

22.1 Overview

- Shadows help viewer to locate objects



- Already seen shadows in ray tracer: how about interactive?
- Will consider three methods:

  Projective shadows, shadow maps, volume shadows

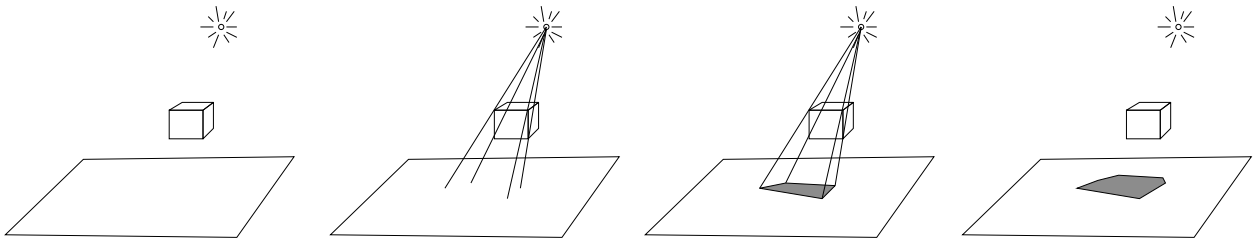
22.2 Projective Shadows

For drawing shadows on a plane.
After drawing scene,

- Draw ray from light through corners of polygon and intersect with plane.
- Draw projected polygon as a dark colour with alpha blending

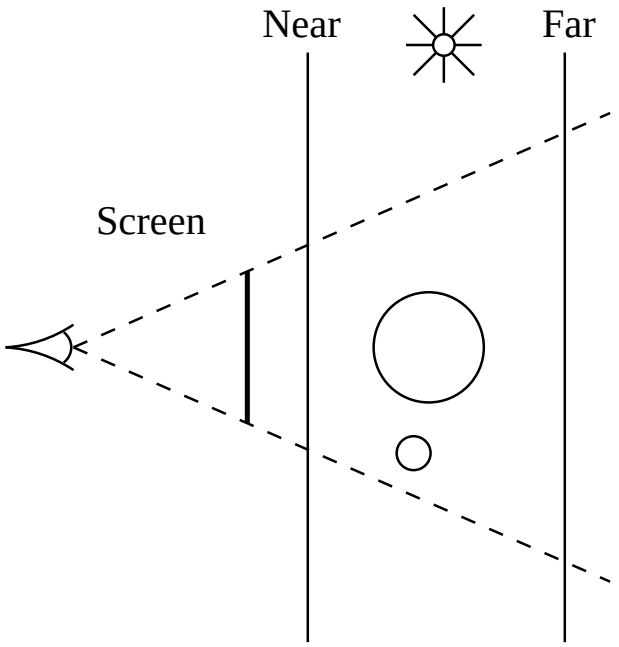
  Depth test strictly less than!

Advantages: fast, simple
Disadvantages: shadows only on a plane

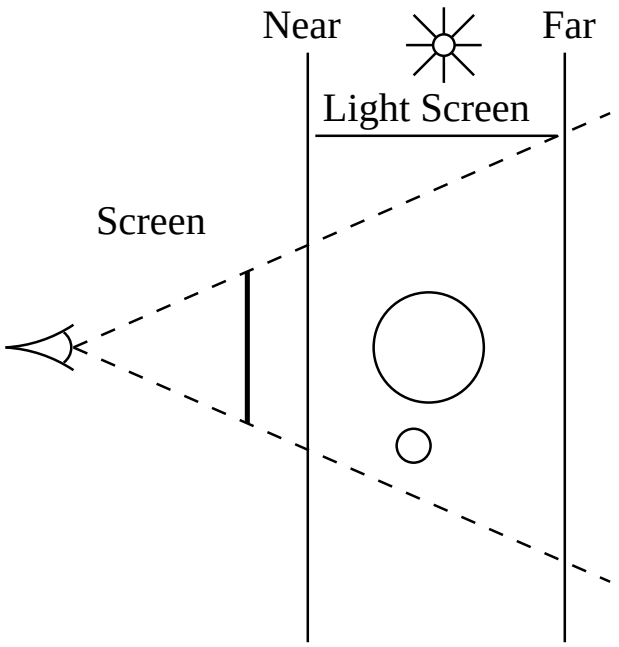


22.3 Shadow Maps

Given a scene...

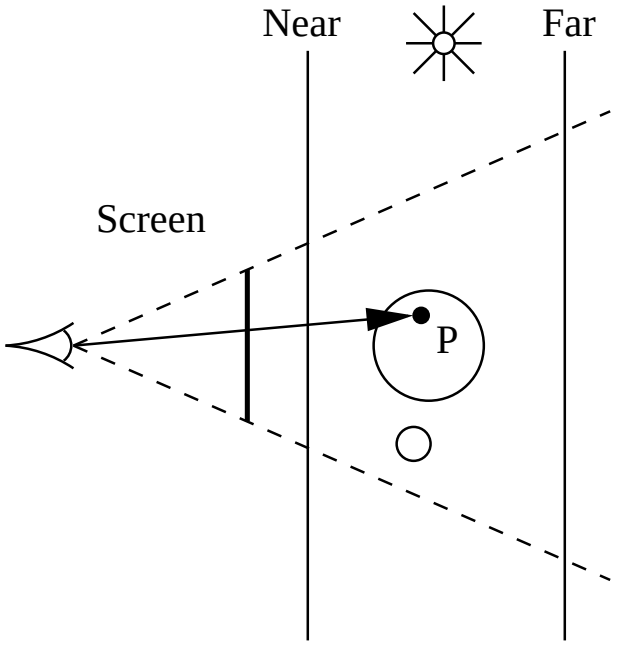


Render scene from *light's* viewpoint (disable colour write)



- Store the z-map (this is the shadow map)
- Possibly render "deeper"

Now render from eye's viewpoint

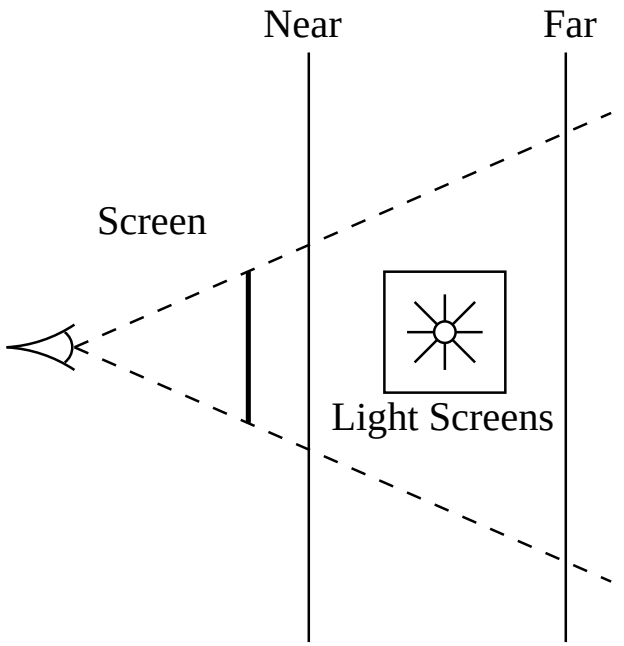


At point P that eye sees on surface,

- Convert P's coordinates to light's frame
- Look up distance from light using shadow map

- If distance in shadow map "equal" to distance from light to P, do lighting calculation. Otherwise in shadow.
- One (or more!) shadow map(s) per light
- Aliasing (lots of work on this)
- Depth test (lots of work on this)
- Needs hardware support
  - Special purpose hardware (obsolete)
  - Use fragment shader to do look up
- If you don't move lights or objects (just view),
  then you don't need to rerender shadow maps each frame.
- Works best with directional and spot lights...

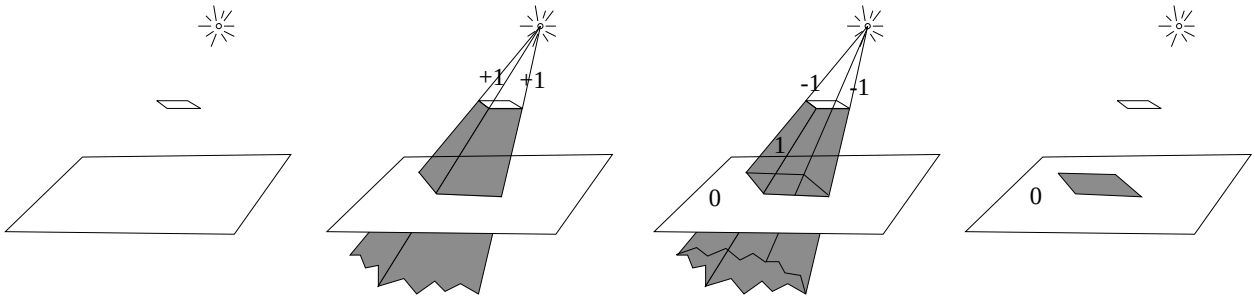
With point light source in viewing volume, have to put cube around light and render 6 shadow maps



22.4 Shadow Volumes

- Multiple drawing passes to render shadows
- Uses stencil buffer to determine where shadows are
  Stencil buffer: store per pixel data, do operations and tests on stencil data, and use to affect drawing to frame buffer

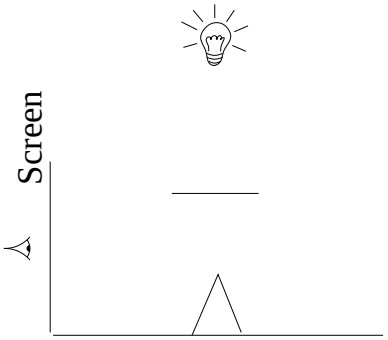
- Idea (assume one light source):
  - For each point P on surface, consider shadow planes cast by silhouette edges
  - Count each front facing shadow plane as +1
    and each rear facing plane as -1
  - If sum is > 0 then P in shadow, otherwise light reaches P.
- We can use graphics hardware for shadow volumes as described on next slide.



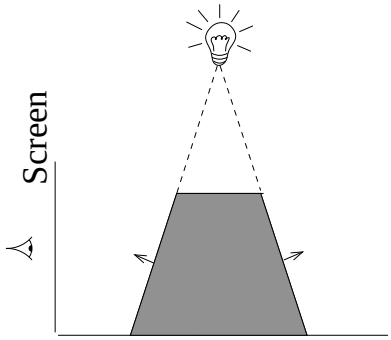
1. Draw scene normally
2. Draw in stencil buffer, front facing shadow polygons using prev depth buffer (do not update depth buffer)

   Increment stencil buffer on draws
3. Draw in stencil buffer, backfacing polygons polygons

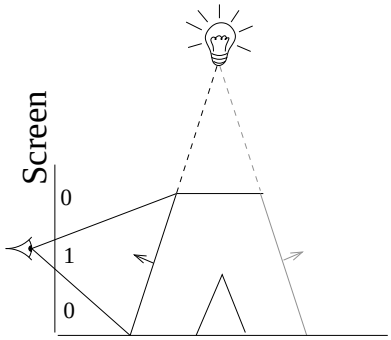
   Decrement stencil buffer on draws
4. Redraw scene with light off, but only update pixels that
   - have exact depth match
   - have non-zero stencil value



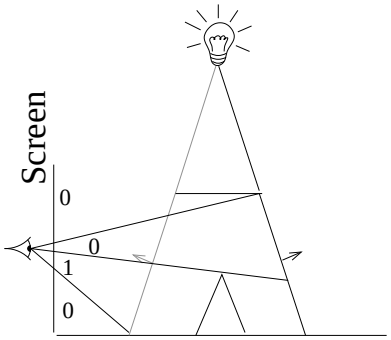
Scene



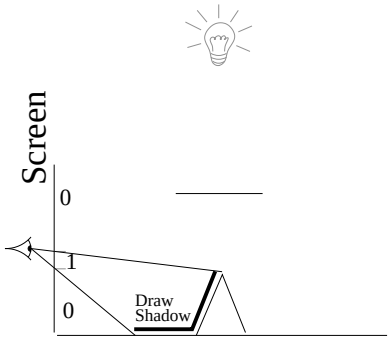
Shadow Volume



Front facing +1

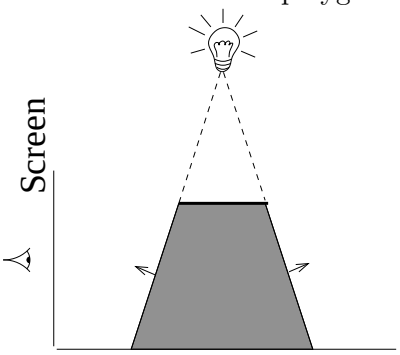


Back facing -1

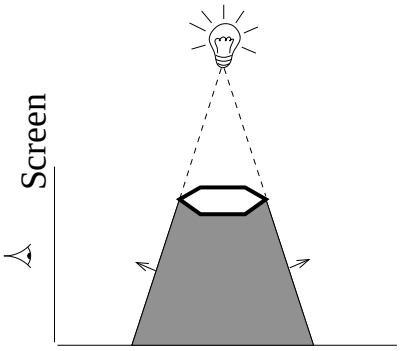


Redraw with light off

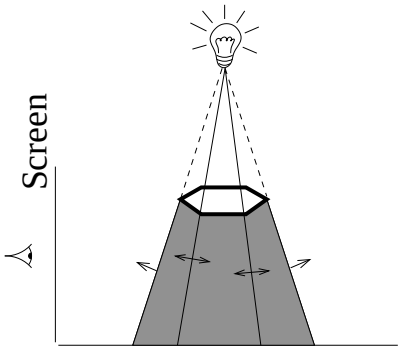
When more than one polygon...



Single Polygon



Silhoutte



Multiple Polygons

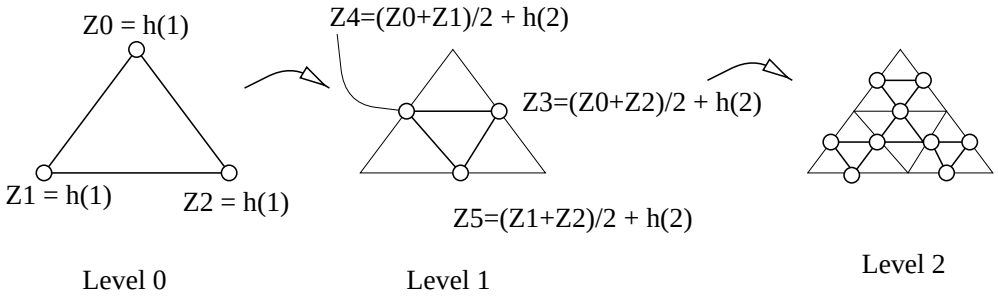
- Draw shadow faces only for silhouette edges of polyhedra
  Problem: have to detect silhouette edges
- Draw shadow volume for each polygon facing away from light
  Shadow face for two polygons sharing an edge cancel
  Only silhouette edges have only one of front/back pair
  Downside: a lot more drawing than needed
- Lots of drawing modes to get right

- Infinite polygons?
  OpenGL tweaks can give you this
- Light map may help
- Problems if viewpoint inside shadow volume
- Need to repeat for each light

23 Modeling of Natural Phenomena

23.1 Fractal Mountains

1. How do we model mountains?
   - Lots of semi-random detail
   - Difficult to model by hand
   - Idea: Let computer generate random effect in controlled manner
2. Start with triangle, refine and adjust vertices
   - At each step of refinement, adjust height of vertices
   - Use scaled random numbers to adjust height



3. Details
   - What to use for h?
     Random number, Gaussian distribution, scaled based on level
     I.e., for deeper level, use smaller scale
   - May want to do first step manually
     Triangular grid, place peaks and valleys
   - Could use different h functions to simulate different effects
   - Automatic colour generation
     high == white
     medium == brown
     low == green
4. Results:
   - Not physically based, but...
   - Look good for little work
   - Don't look at from directly overhead

(Readings: Red book, 9.5; White book, 20.3;
References; Fornier, Fussell, Carpenter, Computer Rendering of Stochastic Models, GIP, 1982
)

23.2 L-system Plants

1. L-systems: Context free plants
2. Idea: Model growth/structure of plant as CFG
3. CFG: Symbols, rules
   - A, B, C, D
   - $A \rightarrow CBA$, $B \rightarrow BD$
4. Start symbol A

$$A \rightarrow CBA \rightarrow CBDCBA \rightarrow CBDDCBDCBA$$

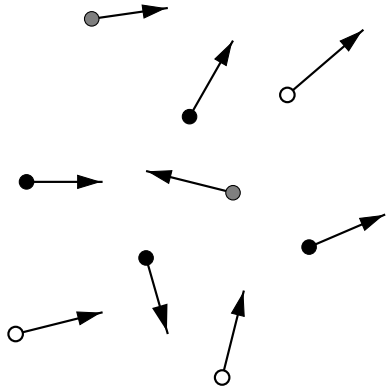
5. Plants need more complex L-systems/geometry
   - Thickness
   - Semi-random, 3D branching
   - Leaf model
6. Top three things to make it look good:
   Texture mapping,
   Texture mapping,
   Texture mapping
7. Plants don't grow like this
   - Grow from top up, outside out
   - Respond to environment (context sensitive)
   - Interact with each other

23.3 Particle Systems

- How do we model "fuzzy" objects? Objects with
  - "soft" boundary
  - changing boundary
  - chaotic behavior

  Examples: Clouds, fire, water, grass, fur
- Could texture map "fuzziness," but changing shape is hard
- Another way is with Particle Systems
  Basic idea: model fuzzy objects as large collection of particles
  Used to model "Genesis" effect in The Wrath of Kahn

- Animate fuzzy or gaseous objects as a changing cloud of "particles"
- Aim for overall effects by having many, simple particles
- Each particle has properties:
  - Geometry (point, sphere, line segment)
  - Position
  - Velocity vector
  - Size
  - Colour
  - Transparency
  - State
  - Lifetime



Animating Particles Systems

- New particles are born, old ones die
- State information describes type of particle
- At each time step, based on state
  - Update attributes of all particles
  - Delete old particles
  - Create new particles
  - Display current state of all particles
- To anti-alias, draw line segment from old position to new position
- To draw grass or fur, draw entire trajectory of particle

Particle System Details

- Easy to implement state machine
- Hard part is determining how to make particles act
- Particles usually independent
  Don't know about one another
- Often will model gravity when updating particles
- Render fire etc using transparency
  Brighter where particle system denser

24 Radiosity

24.1 Definitions and Overview

Radiosity

Radiosity: Diffuse interaction of light

- Ambient term is an approximation to diffuse interaction of light
- Would like a better model of this ambient light
- Idea: "Discretize" environment and determine interaction between each pair of pieces.
- Models ambient light under following assumptions:
  - Conservation of energy in closed environment
  - Only diffuse reflection of light
- All energy emitted or reflected accounted for by its reflection or absorption elsewhere

- All light interactions computed once, in a view independent way
  - Multiple views can then be rendered just using hidden surface removal
  - No mirror/specular reflections
  - Ideal for architectural walkthroughs

Radiance: Electromagnetic *energy flux*, the amount of energy traveling

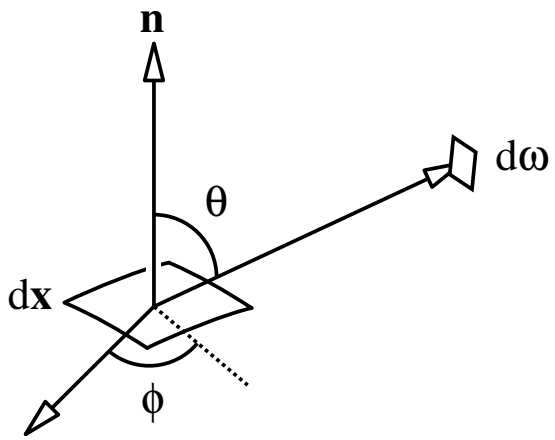
- at some point x
- in a specified direction θ,ϕ
- per unit time
- per unit area perpendicular to the direction
- per unit solid angle
- for a specified wavelength λ
- denoted by $L(x, \theta, \phi, \lambda)$

Spectral Properties: Total energy flux comes from flux at each wavelength

- $L(x, \theta, \phi) = \int_{\lambda_{\min}}^{\lambda_{\max}} L(x, \theta, \phi, \lambda) d\lambda$

Picture: For the indicated situation $L(x, \theta, \phi) dx \cos\theta d\omega dt$ is

- energy radiated through differential solid angle $d\omega = \sin\theta d\theta d\phi$
- through/from differential area dx
- not perpendicular to direction (projected area is $dx \cos\theta$)
- during differential unit time dt



Power: Energy per unit time (as in the picture)

- $L(x, \theta, \phi) dx \cos\theta d\omega$

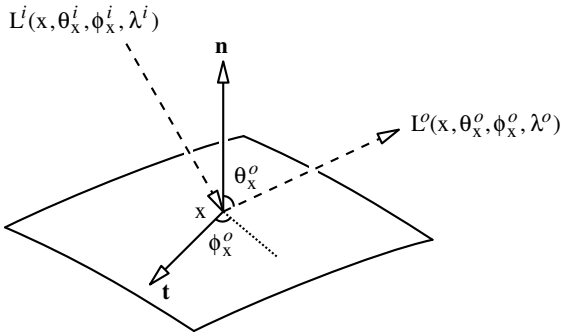
Radiosity: Total power leaving a surface point per unit area

- $\int_\Omega L(x, \theta, \phi) \cos\theta d\omega = \int_0^{\frac{\pi}{2}} \int_0^{2\pi} L(x, \theta, \phi) \cos\theta \sin\theta d\phi d\theta$
  (integral is over the hemisphere above the surface point)

Bidirectional Reflectance Distribution Function:

- is a surface property at a point
- relates energy in to energy out
- depends on incoming and outgoing directions
- varies from wavelength to wavelength
- Definition: Ratio
  - of radiance in the outgoing direction
  - to radiant flux density for the incoming direction

$$\rho_{bd}(x, \theta_i, \phi_i, \lambda_i, \theta_o, \phi_o, \lambda_o) = \frac{L^o(x, \theta_x^o, \phi_x^o, \lambda^o)}{L^i(x, \theta_x^i, \phi_x^i, \lambda^i) \cos\theta_x^i d\omega_x^i}$$



Energy Balance Equation:

$$L^o(x, \quad \theta_x^o, \phi_x^o, \quad \lambda^o) = L^e(x, \theta_x^o, \phi_x^o, \lambda^o) + \int_0^{\frac{\pi}{2}} \int_0^{2\pi} \int_{\lambda_{min}}^{\lambda_{max}} \rho_{bd}(x, \theta_x^i, \phi_x^i, \lambda^i, \theta_x^o, \phi_x^o, \lambda^o) \cos(\theta_x^i) L^i(x, \theta_x^i, \phi_x^i, \lambda^i) d\lambda^i \sin(\theta_x^i) d\phi_x^i d\theta_x^i$$

- $L^o(x, \theta_x^o, \phi_x^o, \lambda^o)$ is the radiance
  - at wavelength λ^o
  - leaving point x
  - in direction θ_x^o, ϕ_x^o
- $L^e(x, \theta_x^o, \phi_x^o, \lambda^o)$: radiance emitted by surface from point
- $L^i(x, \theta_x^i, \phi_x^i, \lambda^i)$: incident radiance impinging on the point
- $\rho_{bd}(x, \theta_x^i, \phi_x^i, \lambda^i, \theta_x^o, \phi_x^o, \lambda^o)$ is the BRDF at the point
  - describes the surface's interaction with light at the point
- the integration is over the hemisphere above the point

Radiosity Approach to Global Illumination:

- Assume that all wavelengths act independently
  - $\rho_{bd}(x, \theta_x^i, \phi_x^i, \lambda^i, \theta_x^o, \phi_x^o, \lambda^o) \equiv \rho_{bd}(x, \theta_x^i, \phi_x^i, \theta_x^o, \phi_x^o)$
  - $L(x, \theta_x, \phi_x, \lambda) \equiv L(x, \theta_x, \phi_x)$
- Assume that all surfaces are purely Lambertian
  - $\rho_{bd}(x, \theta_x^i, \phi_x^i, \theta_x^o, \phi_x^o) \equiv \frac{\rho_d(x)}{\pi}$
- As a result of the Lambertian assumption
  - $L(x, \theta_x, \phi_x) \equiv L(x)$
  - $B(x)$ def. $= \int_\Omega L(x) \cos(\theta_x) d\omega = \pi L(x)$

Simple Energy Balance (Hemisphere Based):

$$L^o(x) = L^e(x) + \int_\Omega \frac{\rho_d(x)}{\pi} L^i(x) \cos(\theta_x^i) d\omega$$

Multiplying by π and letting $E(x) = \pi L^e(x)$:

$$B(x) = E(x) + \rho_d(x) \int_\Omega L^i(x) \cos(\theta_x^i) d\omega$$

But, in general

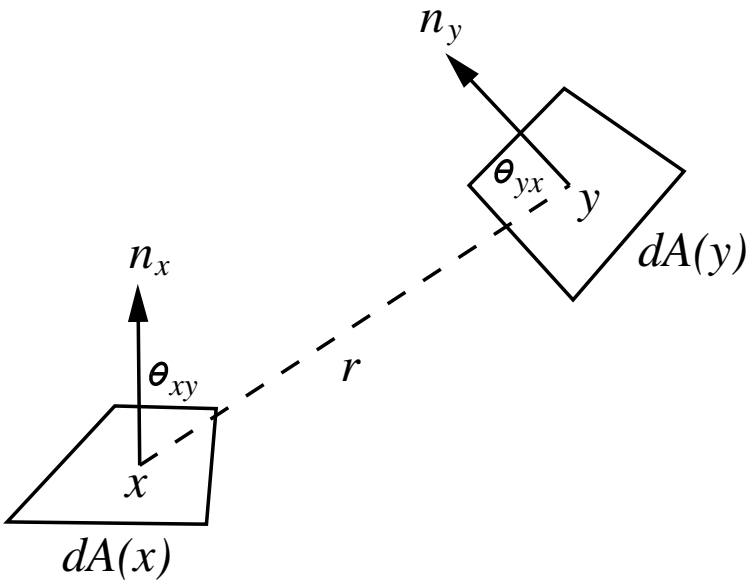
$$L^i(x, \theta_x^i, \phi_x^i) = L^o(y, \theta_y^o, \phi_y^o) \text{ for some } y$$

So we let

$$\theta_x^i = \theta_{xy}, \theta_y^o = \theta_{yx}, \text{ and } L^i(x) = \frac{B(y)}{\pi}$$

for the appropriate point y.

Picture (Scene Based Energy Balance):



Visibility:

We use a term to pick out the special y

$$H(x,y) = \begin{cases} 1 & \text{if } y \text{ is visible from } x \\ 0 & \text{otherwise} \end{cases}$$

Area:

We convert $d\omega$ to surface area

$$d\omega = \frac{\cos\theta_{yx}}{\|x-y\|^2} dA(y)$$

Simple Energy Balance (Scene Based):

$$B(x) = E(x) + \rho_d(x) \int_{y \in \mathcal{S}} B(y) \frac{\cos\theta_{xy}\cos\theta_{yx}}{\pi\|x-y\|^2} H(x,y) dA(y)$$

Piecewise Constant Approximation:

- Approximate the integral by breaking it into a summation over patches
- Assume a constant (average) radiosity on each patch

$$B(y) = B_j \text{ for } y \in P_j$$

$$B(x) \approx E(x) + \rho_d(x) \sum_j B_j \int_{y \in P_j} \frac{\cos\theta_{xy}\cos\theta_{yx}}{\pi\|x-y\|^2} H(x,y) dA(y)$$

- Solve only for a per-patch average density

- $\rho_d(x) \approx \rho_i$ for $x \in P_i$
- $B_i \approx \frac{1}{A_i} \int_{x \in P_i} B(x) dA(x)$
- $E_i \approx \frac{1}{A_i} \int_{x \in P_i} E(x) dA(x)$

Piecewise Constant Radiosity Approximation:

$$B_i = E_i + \rho_i \sum_j B_j \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta_i \cos\theta j}{\pi \|x - y\|^2} H_{ij} dA_j dA_i$$

Form Factor:

$$F_{ij} = \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta_i \cos\theta j}{\pi \|x - y\|^2} H_{ij} dA_j dA_i$$

Note, by symmetry, that we have

$$A_i F_{ij} = A_j F_{ji}$$

Linear Equations: (by Summetry)

$$B_i = E_i + \rho_i \sum_j F_{ij} B_j$$

$$B_i = E_i + \rho_i \sum_j F_{ji} \frac{A_j}{A_i} B_j$$

Radiosity: (Summary)

- Discretize scene into n patches (polygons) each of which emits and reflects light uniformly under its entire area.
- Radiosity emitted by patch i is given by

$$\begin{aligned} B_i &= E_i + \rho_i \sum_j F_{ij} B_j \\ &= E_i + \rho_i \sum_j F_{ji} \frac{A_j}{A_i} B_j \end{aligned}$$

  - $B_i,\ B_j$: radiosity in energy/unit-time/unit-area
  - E_i: light emitted from patch i
  - ρ_i: patch i's reflectivity
  - $F_{j,i}$: Form factor specifying fraction of energy leaving j that reaches i (accounts for shape, orientation, occulsion)
  - $A_i,\ A_j$: Area of patches

Radiosity: Full Matrix Solution

- The equations are

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{ij}$$

or

$$B_i - \rho_i \sum_{1 \le j \le n} B_j F_{ij} = E_i$$

- In matrix form

$$\begin{bmatrix} 1-\rho_1 F_{1,1} & -\rho_1 F_{1,2} & \dots & -\rho_1 F_{1,n} \\ -\rho_2 F_{2,1} & 1-\rho_2 F_{2,2} & \dots & -\rho_2 F_{2,n} \\ \vdots & & \ddots & \vdots \\ -\rho_n F_{n,1} & -\rho_n F_{n,2} & \dots & 1-\rho_n F_{n,n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

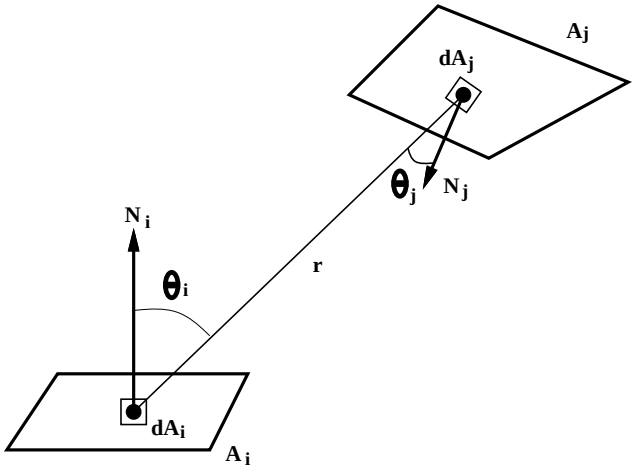
- B_i's are only unknowns
- If $F_{i,i} = 0$ (true for polygonal patches) then diagonal is 1
- Solve 3 times to get RGB values of B_i

(Readings: Hearn and Baker in OpenGL: 10-12. Watt: Chapter 11.)

24.2 Form Factors

Calculation

- Form factor specifies fraction of energy leaving one patch that (directly) arrives at another patch.



- The differential form factor is given by

$$ddF_{di,dj} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\ dA_j\ dA_i$$

where

$$H_{ij} = \begin{cases} 0 & \text{if } dA_i \text{ occluded from } dA_j \\ 1 & \text{if } dA_i \text{ visible from } dA_j \end{cases}$$

Form Factors: Calculation

- To see how much of dA_i illuminates patch j, we integrate:

$$dF_{di,j} = \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij}\ dA_j\ dA_i$$

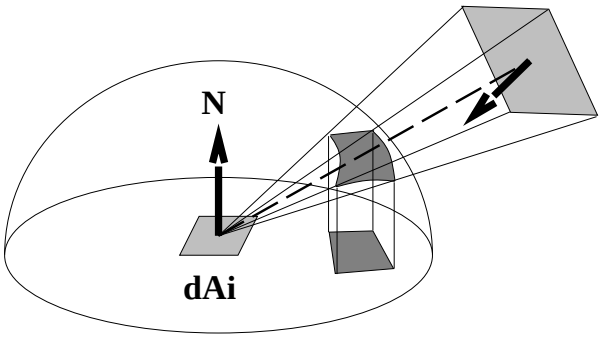
- The form factor from A_i to A_j is an area average of the integral over A_i:

$$F_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} H_{ij} \; dA_j \; dA_i$$

- Typically we approximate this integral:
  - Ray tracing
  - Hemi-sphere
  - Hemi-cube
  - Compute $dF_{di,j}$
- Lots of $F_{i,j}$'s may be zero leading to a sparse matrix

Form Factors: Hemi-sphere Form Factors

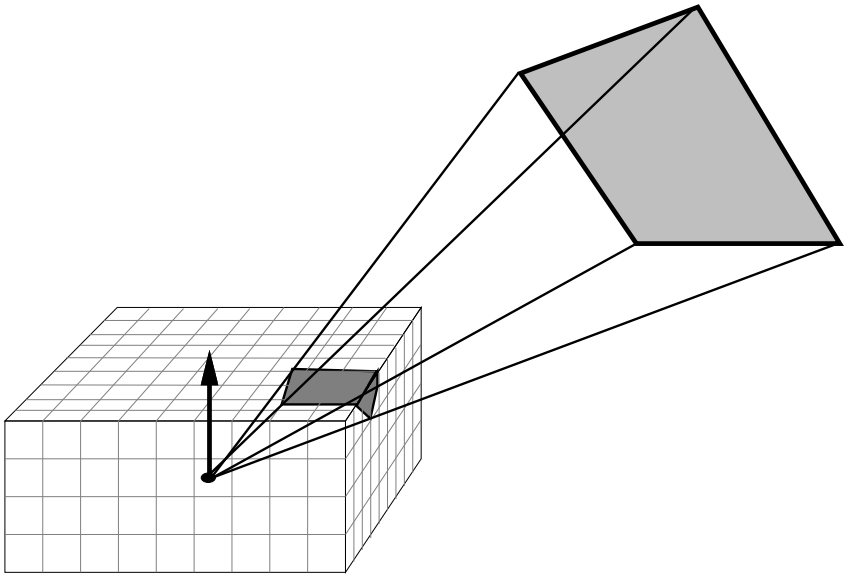
- Place a hemisphere around patch i and project patch j onto it



- Select a point on each patch and look at angles
- Project patch j onto hemisphere, and project projection into circle: $F_{di,j} = \frac{P_A}{H_A}$ where H_A is the area of the circle.
- Note that any polygon with same "cross section" has same form factor
- Projecting onto hemisphere is still hard

Form Factors: Hemi-cube Form Factors

- Project onto a hemi-cube
- Subdivide faces into small squares

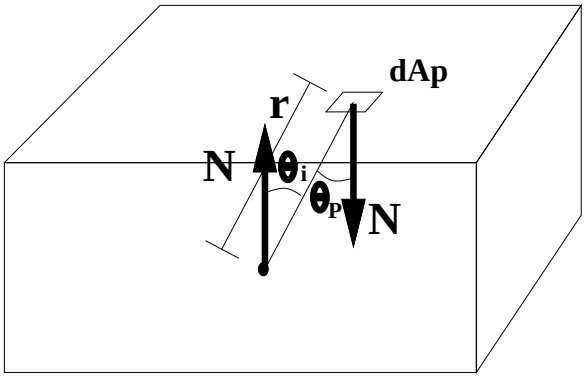


- Now determine which squares patch j projects onto

Form Factors: Delta Form Factors

- Each hemi-cube cell P has pre-computed delta form factor

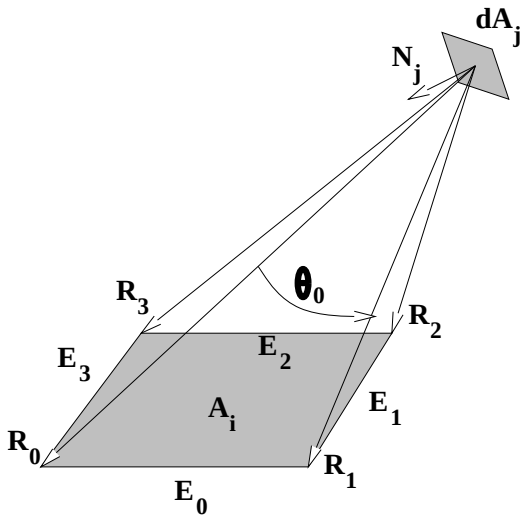
$$\Delta F_P = \frac{\cos(\theta_i)cos(\theta_P)}{\pi r^2}\Delta A_P$$



- Approximate $dF_{dj,i}$ by summing delta form factors
- If distance from i to j is large, this is good approximation for $F_{j,i}$.

Form Factors: Exact Form Factors

- We can compute the exact form factor for point to polygon



The differential form factor is given by $F_{dA_j A_i} = \frac{1}{2\pi} \sum_i N_j \cdot \Gamma_i$ where

- N_j is the normal to patch j
- Γ_i is the vector in direction $R_{i+1} \times R_i$ with length equal to θ_i (in radians).

- Doesn't account for occlusion

24.3 Progressive Refinement

- Recall our equation says how much radiosity is comes from patch j to patch i:

$$B_i \text{ due to } B_j = \rho_i B_j F_{ji}$$

Light is being *gathered* at patch i.

- Instead we can ask "how much radiosity" is *shot* from patch i to patch j?

$$\begin{aligned} B_j \text{ due to } B_i &= \rho_j B_i F_{ij} \\ &= \rho_j B_i F_{ji} \frac{A_j}{A_i} \end{aligned}$$

- Idea: Choose a patch, shoot its radiosity to all other patches, repeat.

Progressive Refinement: Code

Keep track of total radiosity (B) and unshot radiosity (dB)

```
Procedure Shoot(i)
For all j, calculate all Fji
Foreach j
    drad = pj*dBi*Fji*Ai/Aj
    dBj += drad
    Bj += drad
endfor
dBi = 0
```

Call from

```
while (unshot > eps) do
    Choose patch i with highest dB
    Shoot(i)
end
```

Progressive Refinement: Analysis

- Shoot lights first
- Can stop and look at partial results
- Can be formulated as a matrix equation
- *Lots* of cute tricks:
  - Add initial ambient to all patches so that initial results more viewable
  - Can move objects and lights after calculation has started
- Technically it is slower than full matrix methods, but...
- Form factors usually not stored since they are computed on the fly
  - Advantage: save storage space
  - Disadvantage: some form factors may need to be computed multiple times

Radiosity Issues

Questions:

- How big should we make initial patches?
- When do we stop Progressive-Refinement?

Problems

- T-vertices and edges cause light/shadow leakage
- Occlusions and shadows hard to model
- Form factor estimations often cause aliasing

Extensions

- Methods for adding specular (but *expensive*)
- Substructuring improves quality without increasing big-O
- Smarter substructuring reduces big-O

24.4 Meshing in Radiosity

Issues

- Initial Mesh Size?
- T-vertices, etc?
- Shadow Boundaries

- Soft Shadows

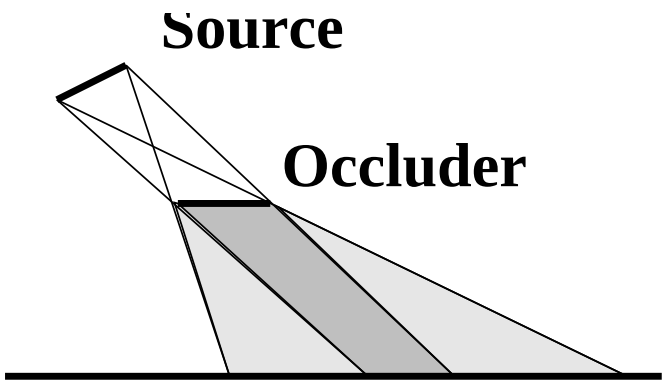
Two Techniques:

- Split along shadow boundaries
- Adaptive Subdivision

Split Along Shadow Boundaries – 2D

3 regions:

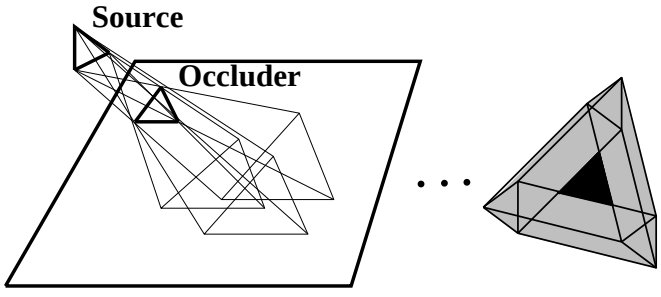
- Lit
- Unlit
- Linear



Split Along Shadow Boundaries – 3D

Lots of regions:

- Lit
- unlit
- Linear
- Quadratic



Adaptive Subdivision

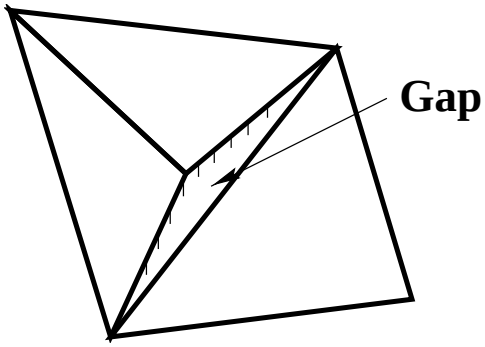
- Make initial Meshing Guess
- Point sample a patch
- Average and check variance

  - Low ⇒ Good estimate
  - High ⇒ Poor estimate
    Subdivide and repeat

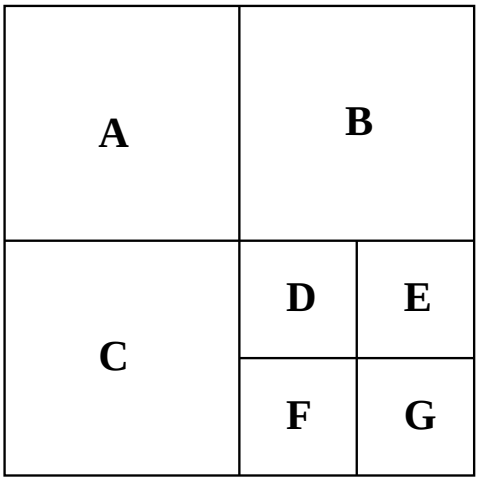
Problems with Adaptive Subdivision

- Radiosity has $O(n^2)$ run time
  Subdivision increases n
- Ideas:
  - Shoot from big patches
  - Receive at small patches

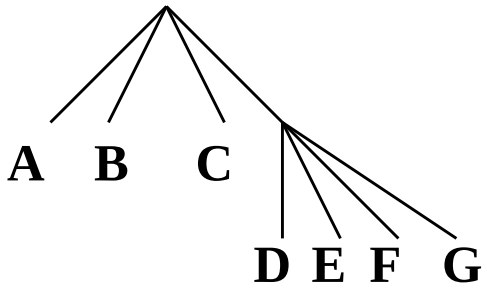
  Sub problems:
  - sub-patch to big-patch association
  - T-vertices



Quad Tree



- Store as

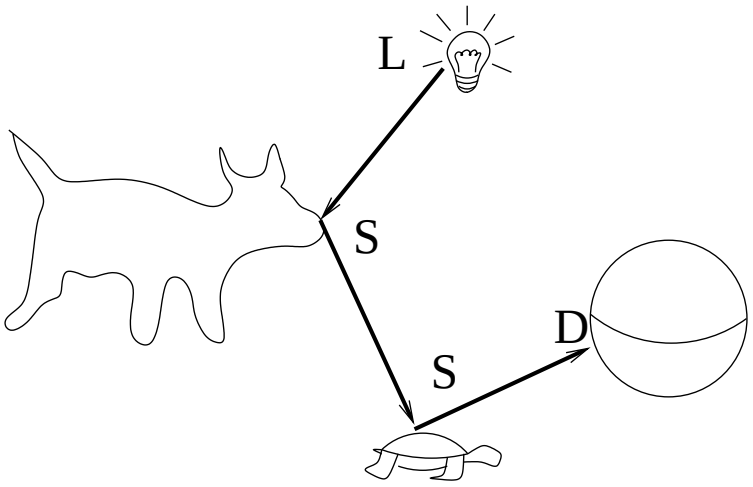


- Traverse up to find parent
- Traverse up-down to find neighbor
- Anchoring

25 Photon Maps

25.1 Overview

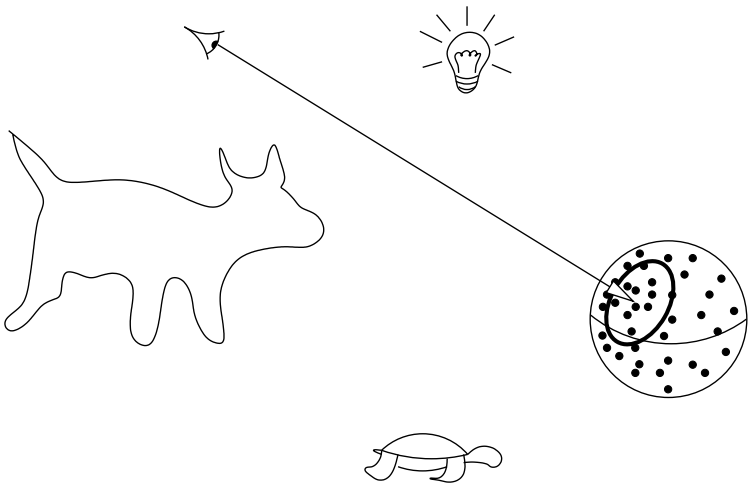
- Radiosity too expensive, doesn't do caustics, etc.
- Photon mapping idea:
  - trace rays from lights, bounce off shiny surfaces,
  - store light information where light hits diffuse surfaces.
  - Then ray trace, and use stored light to achieve global illumination effects.
- General steps of photon mapping:
  - Emit photons
    Mostly distribution of n photons around light source
  - Trace photons
    Bounce off shiny surfaces, store at diffuse surfaces
  - Estimate irradiance
  - Ray trace, using estimated irradiance as needed
- LS+D: Idea based on bouncing light off shiny surfaces until it hits diffuse surface



- L(S|D)*D: bounce off both shiny and diffuse surfaces and store information at diffuse surfaces.
- "Shiny" surface: light leaves in a fixed direction (relative to incoming direction)
  Includes transparent (refractive) surfaces
- Similar to ray tracing, but cast rays from lights
- When light hits shiny surface, bounce to next surface
  Use Russian Roulette to send ray to next surface based on BRDF.
- When light hits diffuse surface, store surface point in data structure (kD tree, etc).
  Also store surface normal, intensity/power of light.

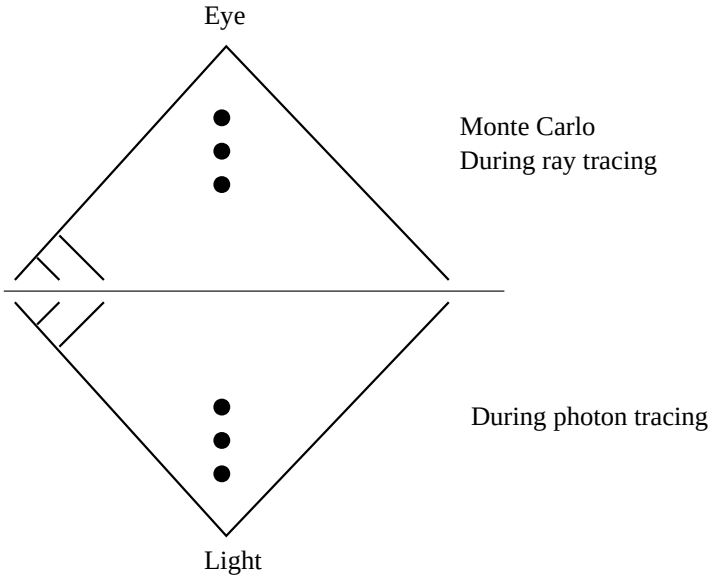
To estimate irradiance at a point,

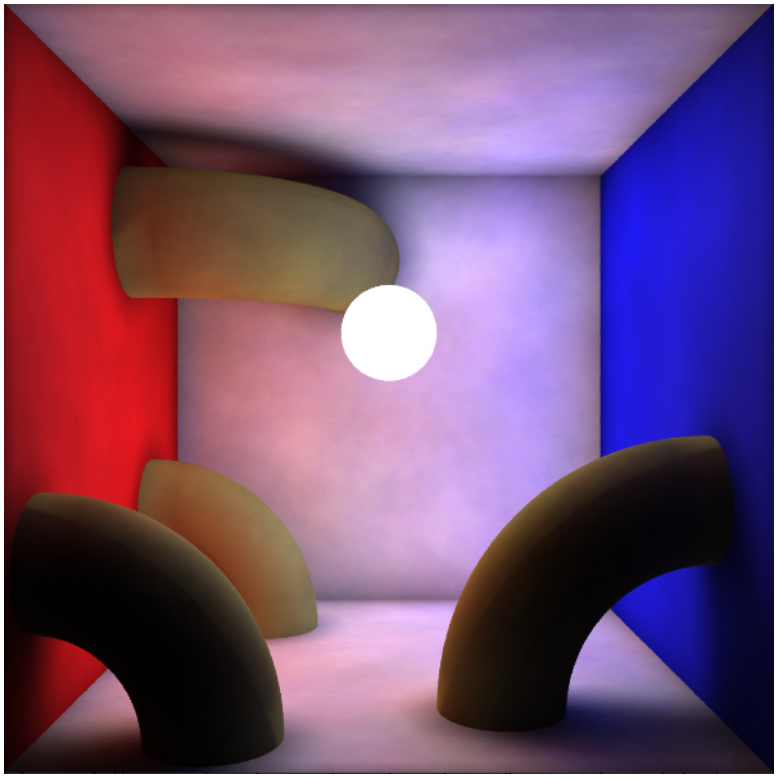
- Do a k nearest neighbour search to find closest stored photon information
  Usually try to limit search to a local region (sphere)



- Average to get irradiance for the point
- Since kD-tree is space (not surface) based, may pick up photons on nearby surface
  May cause noise in image
- When ray tracing, use radiance information in lighting equation
- To get caustics, use
  Direct light + mirror + refraction + ambient + irradiance est
- To use for global illumination, use
  Direct light + mirror + refraction + irradiance est
  but photon tracing is performed differently

- Need to work harder or smarter to get global illumination
- Harder: trace a LOT more rays from light
- Smarter: trace rays from light (Russian Roulette)
  and trace rays from eye (Monte Carlo)
  and meet in the middle





- Number of photons to cast from light
- Parameters in irradiance estimate
  value of k, radius of limiting volumes, etc.
- Monte Carlo vs Russian Roulette
- Noise, noise, noise

26 Polyhedral Data Structures

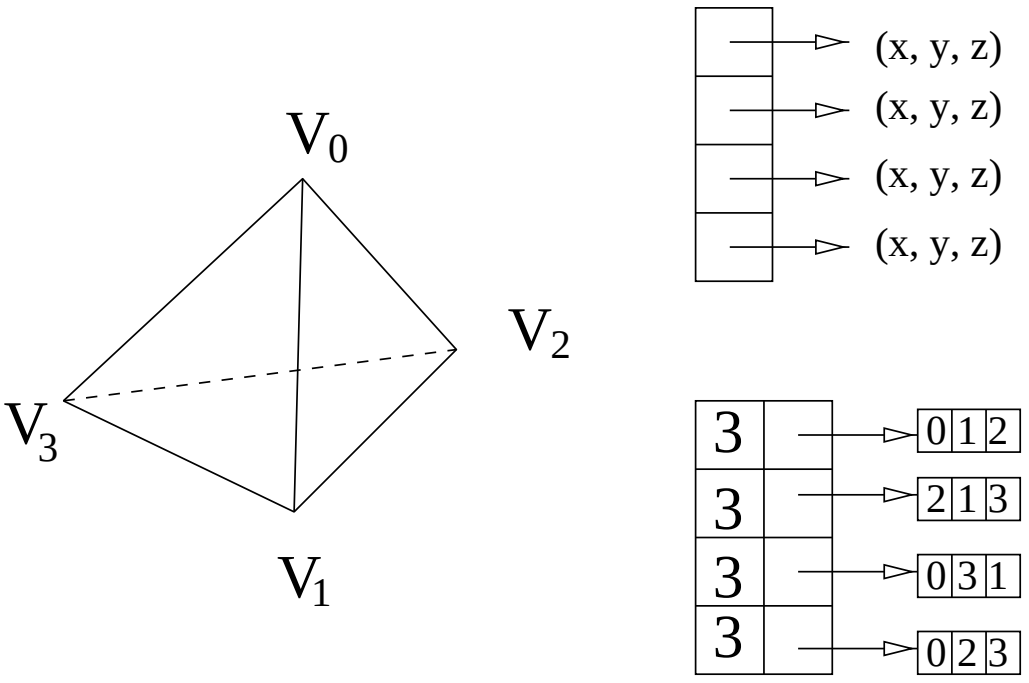
26.1 Storage Models

Polyhedra

- How do we store a polyhedron in memory?
  Must consider the following:
  - Memory
  - Efficiency
    Operations
- Will look at three formats
  - Ray Tracer
  - Generalized Ray Tracer
  - Winged-edge variations

Ray Tracer Format

- We already saw one ASCII format (for the ray tracer)
- Data structure for this is simple:
  - Array of points (triples of floats)
  - Array of faces, with pointers to points

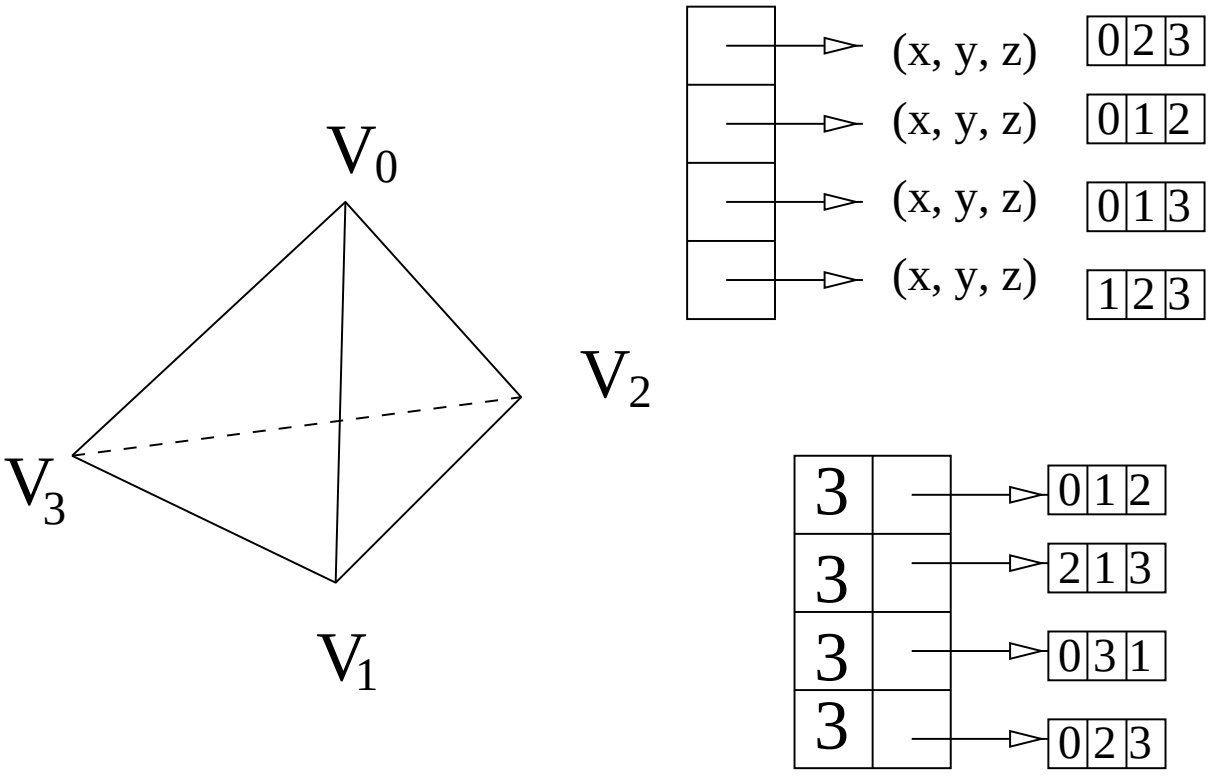


Compact, simple to get face information

- Advantages
  - Space efficient
  - Easy to generate, parse, build data structure
- Disadvantages
  - Vertex neighbours?
  - Modify the data structure?

  Example: What if we want to know faces surround a vertex?

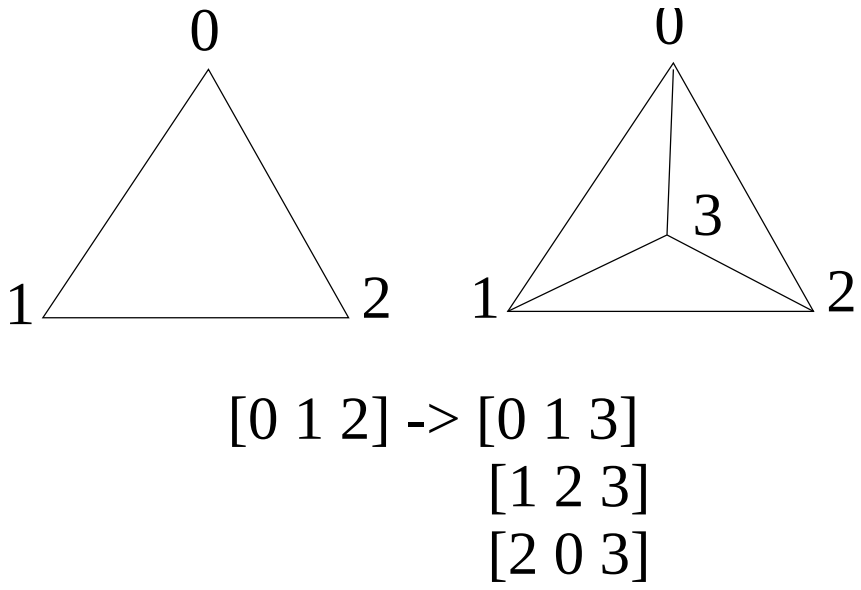
  Radiosity might want this
- Solution: For each point, have a list of pointers to surrounding faces.
  - Can use same ASCII input format
  - Build lists when reading data



Question: Where do these lists come from?

- Problems still remain:
  - How do we modify a polyhedron?
  - Easy to move vertex

- Add face or vertex is easy
- Split face not too hard



- Delete is hard
  $O(n)$ or leaves holes in our list

(Readings: Hearn and Baker in OpenGL: 8-1 (not very useful). Watt: 2.1.)

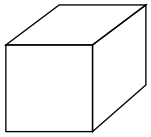
26.2 Euler's Formula

- Closed polyhedron:

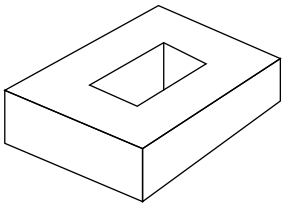
$$V - E + F - H = 2(C - G)$$

  - V: Number of vertices
  - E: Number of edges
  - F: Number of faces
  - H: Number of holes
  - C: Number of connected components
  - G: Number of genus (sphere = 0, torus = 1)

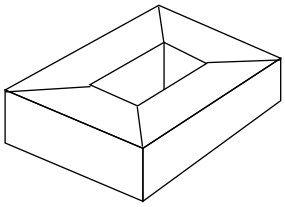
- Examples:

Cube 

V= E= F= G= C= H=

Torus 1 

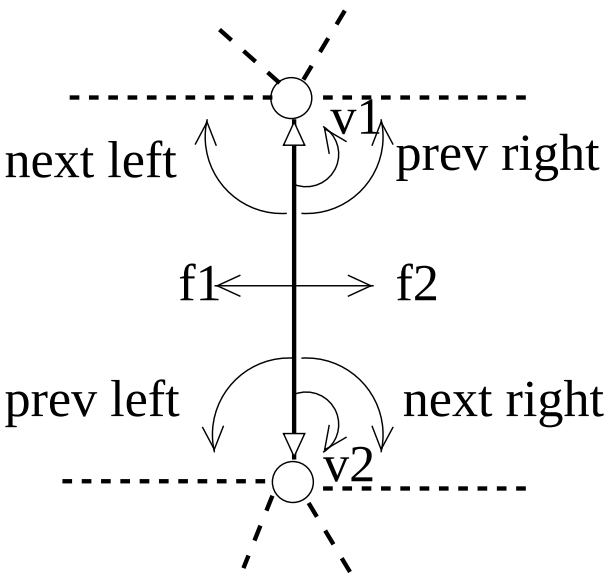
V= E= F= G= C= H=

Torus 2 

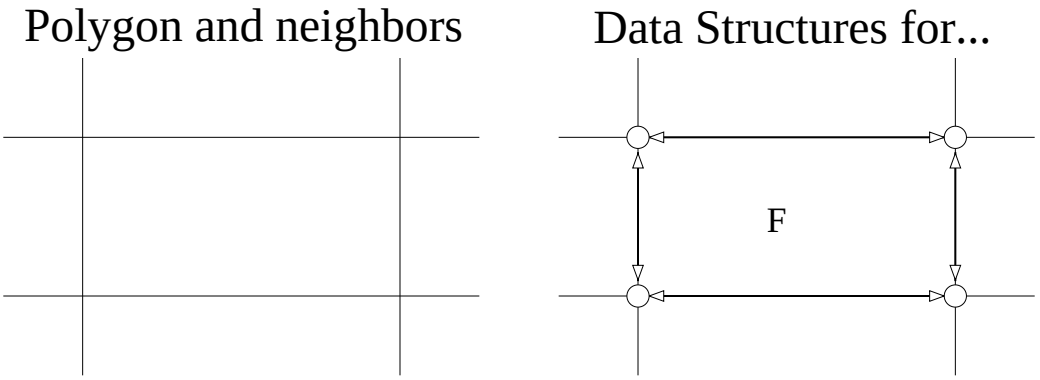
V= E= F= G= C= H=

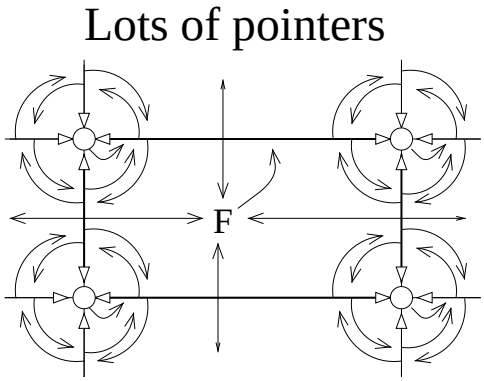
26.3 Winged Edge

- Operations:
  - Want to traverse
    - Neighbouring faces of a face/vertex
    - Neighbouring vertices of a face/vertex
    - Edges of a face
  - Want to modify (add/delete) the polyhedron
- Key ideas of winged-edge:
  - Edge is important topological data structure
  - Also, separate topology from geometry
- Edge is the primary data structure



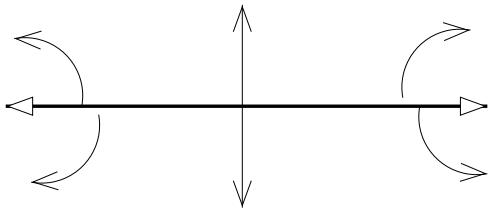
- Pictorially:



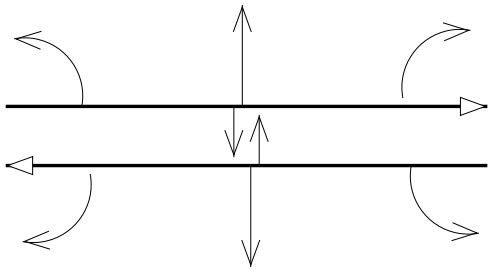


Variations

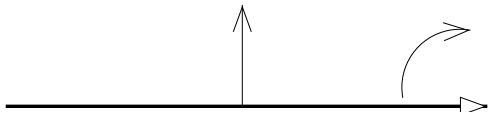
- Some operations simplified by splitting edge into half-edges
- Uses a lot of space. Time-space trade-off.



Winged Edge



Half-edge



Minimal Edge

Data types

```
struct he {
  struct he* sym;
  struct he* next;
  struct he* prev;
  struct vert* v;
  struct face* f;
  void* data;
}
struct vert {
  struct he* rep;
  void* data;
}
```

```
struct face {
  struct he* rep;
  void* data;
}
```

- Example: Neighbouring faces of `f`

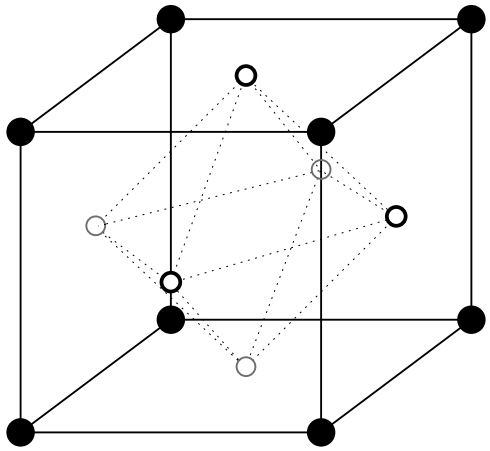
```
struct face* f;
struct he* e;
struct he* s;
struct face* fl[LOTS];
int i;

s = e = f->rep;
i = 0;
```

```
do {
  fl[i++] = e->sym->f;
  e = e->next;
} while (e != s);
```

- Example: Removing an edge

```
RemoveEdge(struct he* e) {
  /* fix edge->edge pointers */
  e->prev->next = e->sym->next;
  e->next->prev = e->sym->prev;
  e->sym->next->prev = e->prev;
  e->sym->prev->next = e->next;
  /* fix edge->face pointers */
  ee = e->sym->next;
  while (ee != e->sym) {
    ee->f = e->f;
    ee = ee->next;
  }
  /* fix face->edge pointers */
  e->f->rep = e->prev;
  /* fix vertex->edge pointers */
  e->v->rep = e->sym->prev;
  e->sym->v->rep = e->prev;
  DeleteFace(e->sym->f);
  DeleteEdge(e);
}
```

- Manifolds with Boundary
  - Half-edge allows representation of manifold w/boundary
    Set sym pointer to NULL
  - Allows for lone face
  - Makes some constructions easier
- Duality
  - Faces and vertices are dual
    Representations and rôles are identical
  - Can replace each face with a vertex and each vertex with a face



  - If boundaries, then dual may not be manifold

- Evaluation

  \+ Faces with arbitrary number of sides
  Vertices with arbitrary number of neighbours

  \+ Iterate over everything

  – Not space efficient

  – Painful to write/debug code

  ** Simpler data structure better unless you need to modify, iterate

- API

  - Discussions in papers use low level operators
  - Better if user of package does NOT manipulate pointers
  - Higher level "safe" operators, iterators:
    * `ForeachMeshVertex`
    * `ForeachFaceVertex`
    * `Split 3-1`
    * `RemoveEdge`
  - Example: Split all faces 3-1

```
ForeachMeshFace(m,f) {
  p = SplitFace3to1(f);
  SetPosition(P,x,y,z);
}
```

  - Which operations should we provide?

27 Splines

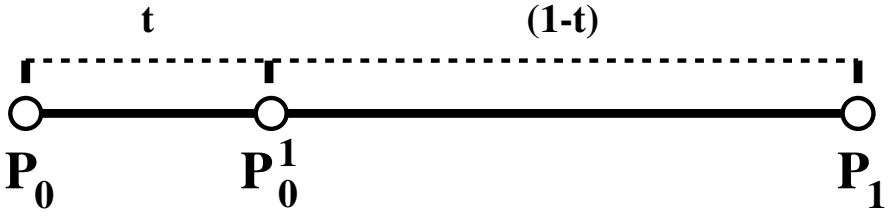
27.1 Constructing Curve Segments

Constructing Curve Segments

Linear blend:

- Line segment from an affine combination of points

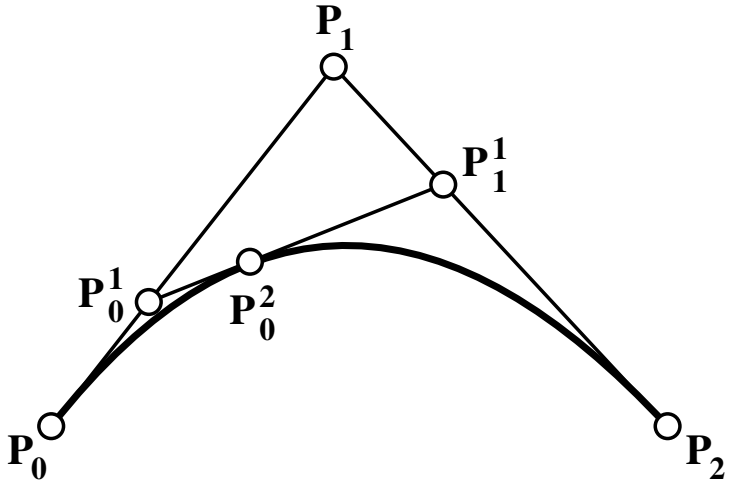
$$P_0^1(t) = (1-t)P_0 + tP_1$$



Quadratic blend:

- Quadratic segment from an affine combination of line segments

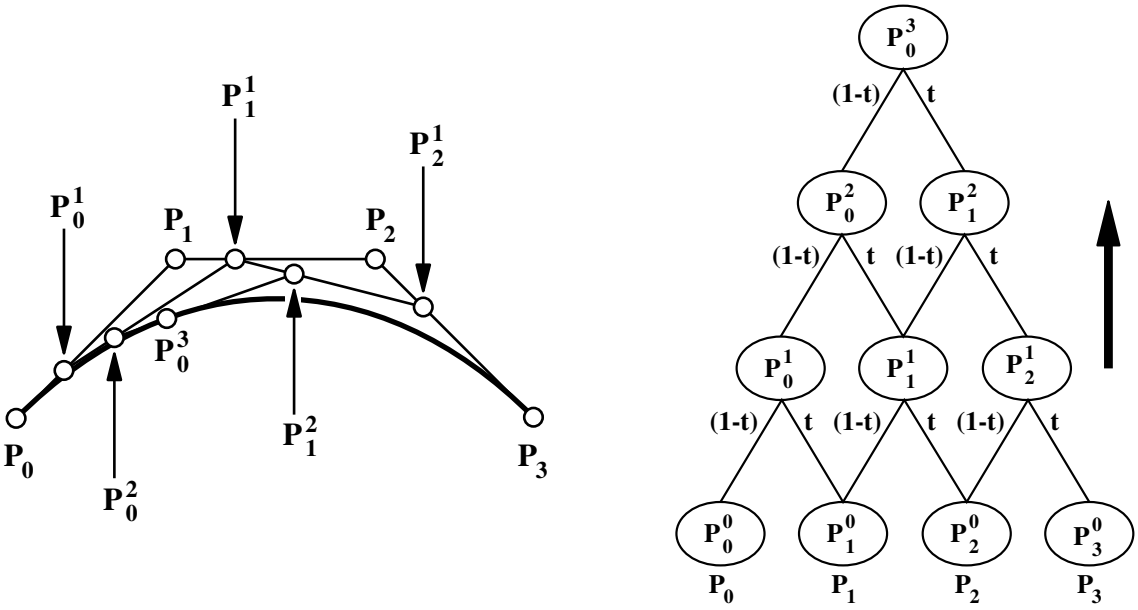
$$\begin{aligned} P_0^1(t) &= (1-t)P_0 + tP_1 \\ P_1^1(t) &= (1-t)P_1 + tP_2 \\ P_0^2(t) &= (1-t)P_0^1(t) + tP_1^1(t) \end{aligned}$$



Cubic blend:

- Cubic segment from an affine combination of quadratic segments

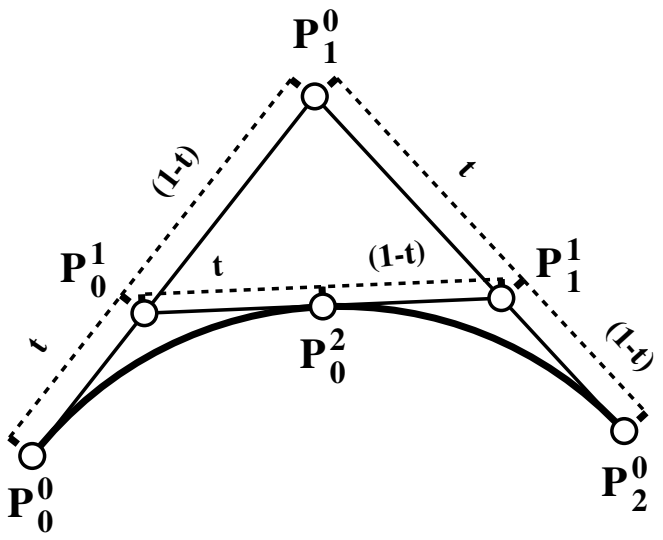
$$\begin{aligned}
P_0^1(t) &= (1-t)P_0 + tP_1 \\
P_1^1(t) &= (1-t)P_1 + tP_2 \\
P_0^2(t) &= (1-t)P_0^1(t) + tP_1^1(t) \\
P_1^1(t) &= (1-t)P_1 + tP_2 \\
P_2^1(t) &= (1-t)P_2 + tP_3 \\
P_1^2(t) &= (1-t)P_1^1(t) + tP_2^1(t) \\
P_0^3(t) &= (1-t)P_0^2(t) + tP_1^2(t)
\end{aligned}$$



- The pattern should be evident for higher degrees.

Geometric view (deCasteljau Algorithm):

- Join the points P_i by line segments
- Join the $t : (1-t)$ points of those line segments by line segments
- Repeat as necessary
- The $t : (1-t)$ point on the final line segment is a point on the curve
- The final line segment is tangent to the curve at t



Expanding Terms (Basis Polynomials):

- The original points appear as coefficients of *Bernstein polynomials*

$$\begin{aligned} P_0^0(t) &= 1 \cdot P_0 \\ P_0^1(t) &= (1-t)P_0 + tP_1 \\ P_0^2(t) &= (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \\ P_0^3(t) &= (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3 \end{aligned}$$

$$P_0^n(t) = \sum_{i=0}^{n} P_i B_i^n(t)$$

$$\text{where} \qquad B_i^n(t) = \tfrac{n!}{(n-i)!i!}(1-t)^{n-i}t^i = \left(\begin{array}{c} n \\ i \end{array} \right)(1-t)^{n-i}t^i$$

- The Bernstein polynomials of degree n form a basis for the space of all degree-n polynomials

27.2 Bernstein Polynomials

Bernstein Polynomial Properties

Partition of Unity: $\sum_{i=0}^{n} B_i^n(t) = 1$

Proof:

$$\begin{aligned} 1 &= (t + (1-t))^n \\ &= \sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right)(1-t)^{n-i}t^i \\ &= \sum_{i=0}^{n} B_i^n(t). \end{aligned}$$

Non-negativity: $B_i^n(t) \geq 0, \text{for } t \in [0,1]$.

Proof:

$$\begin{aligned} \binom{n}{i} &> 0 \\ t &\geq 0 \text{ for } 0 \leq t \leq 1 \\ (1-t) &\geq 0 \text{ for } 0 \leq t \leq 1 \end{aligned}$$

Many other nice properties...

27.3 Bézier Splines

Bezier Curves

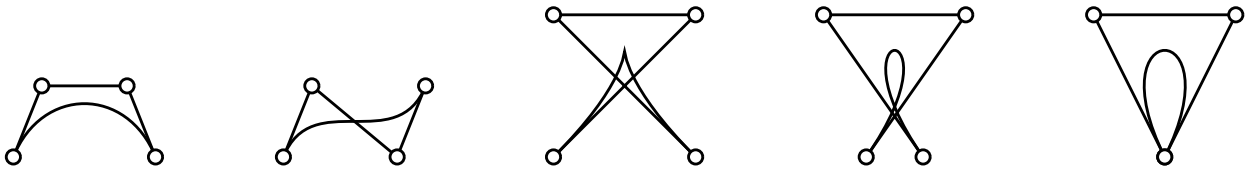
Definition:

A degree n (order $n+1$) **Bézier curve segment** is

$$P(t) = \sum_{i=0}^{n} P_i B_i^n(t)$$

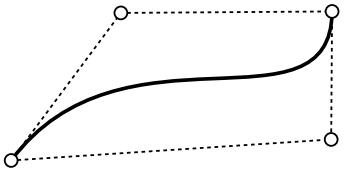
where the P_i are k-dimensional **control points**.

Examples:



Convex Hull:

$\sum_{i=0}^{n} B_i^n(t) = 1$, $B_i^n(t) \geq 0$ for $t \in [0,1]$
$\Longrightarrow P(t)$ is a convex combination of the P_i for $t \in [0,1]$
$\Longrightarrow P(t)$ lies within convex hull of P_i for $t \in [0,1]$.



Affine Invariance:

- A Bézier curve is an affine combination of its control points.

- Any affine transformation of a curve is the curve of the transformed control points.

$$T\left(\sum_{i=0}^{n} P_i B_i^n(t)\right) = \sum_{i=0}^{n} T\left(P_i\right) B_i^n(t)$$

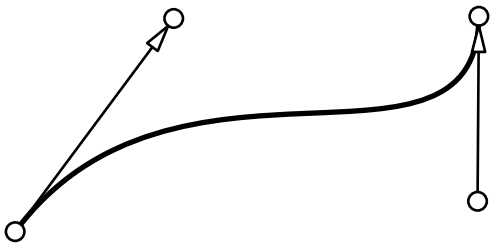
- **This property does not hold for projective transformations!**

Interpolation:

$B_0^n(0) = 1$, $B_n^n(1) = 1$, $\sum_{i=0}^{n} B_i^n(t) = 1$, $B_i^n(t) \geq 0$ for $t \in [0,1]$
$\Longrightarrow B_i^n(0) = 0$ if $i \neq 0$, $B_i^n(1) = 0$ if $i \neq n$
$\Longrightarrow P(0) = P_0$, $P(1) = P_n$.

Derivatives:

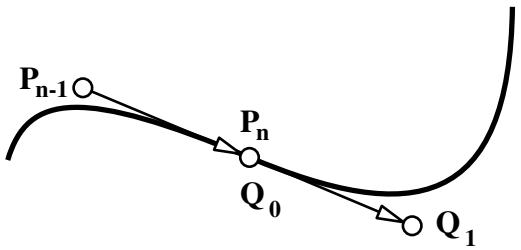
$\frac{d}{dt} B_i^n(t) = n(B_{i-1}^{n-1}(t) - B_i^{n-1}(t))$
$\Longrightarrow P'(0) = n(P_1 - P_0)$, $P'(1) = n(P_n - P_{n-1})$.



Smoothly Joined Segments ($\mathbf{C^1}$): :

Let P_{n-1}, P_n be the last two control points of one segment.
Let Q_0, Q_1 be the first two control points of the next segment.

$$\begin{aligned} P_n &= Q_0 \\ (P_n - P_{n-1}) &= (Q_1 - Q_0) \end{aligned}$$



Smoothly Joined Segments ($\mathbf{G^1}$):

$$\begin{aligned} P_n &= Q_0 \\ (P_n - P_{n-1}) &= \beta(Q_1 - Q_0) \text{ for some } \beta > 0 \end{aligned}$$

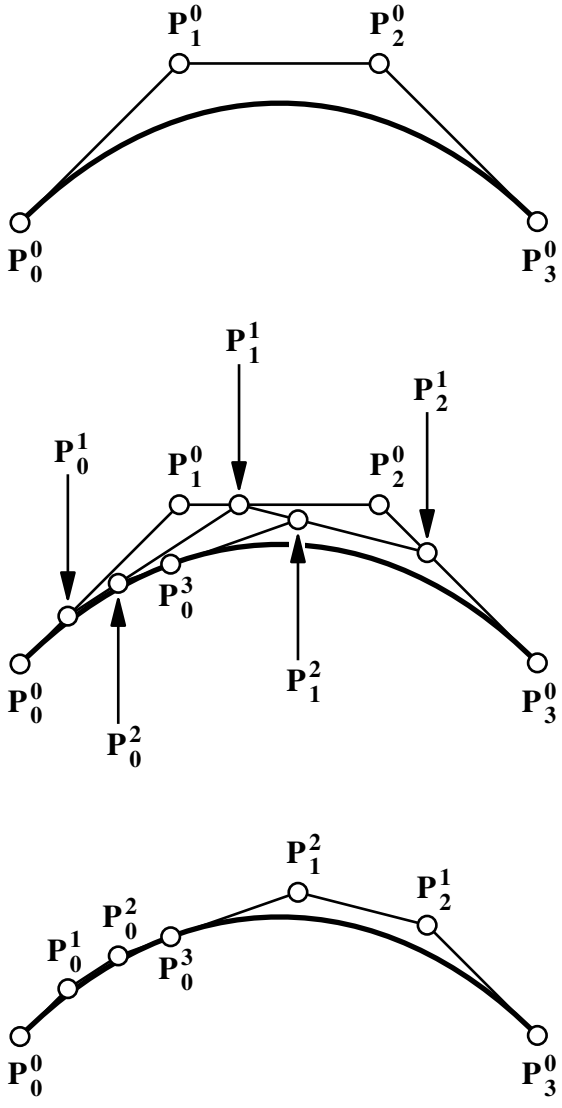
Recurrence, Subdivision:

$B_i^n(t) = (1-t)B_i^{n-1} + tB_{i-1}^{n-1}(t)$
$\Longrightarrow$ de Casteljau's algorithm:

$$\begin{aligned} P(t) &= P_0^n(t) \\ P_i^k(t) &= (1-t)P_i^{k-1}(t) + tP_{i+1}^{k-1} \\ P_i^0(t) &= P_i \end{aligned}$$

Use to evaluate point at t, or subdivide into two new curves:

- P_0^0, P_0^1, $\ldots P_0^n$ are the control points for the left half;
- P_n^0, P_{n-1}^1, $\ldots P_0^n$ are the control points for the right half



27.4 Spline Continuity

Polynomials Inadequate

- Weierstrass Approximation Theorem: Can approximate any C^0 curve to any tolerance with polynomials

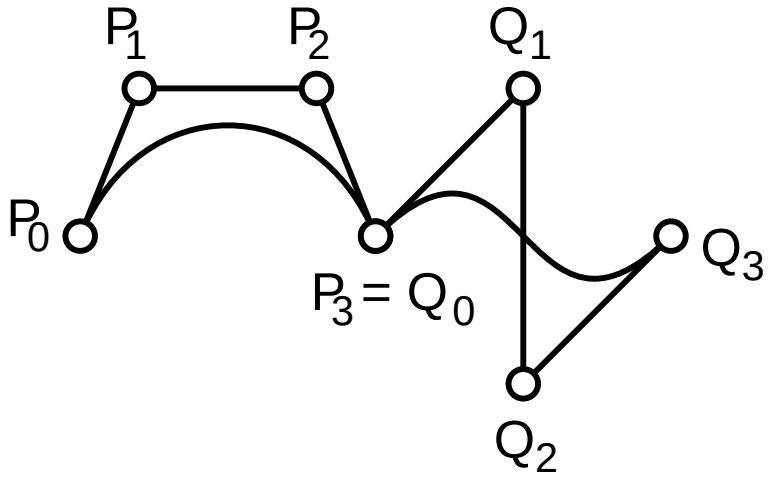
  But may require high degree
- To model complex curves, will need high degree
- High degree:
  - Non-local
  - Expensive to evaluate
  - Slow convergence as we increase degree

Piecewise Polynomials

- Idea: Instead of one high degree polynomial, piece together lots of low degree polynomials
- Benefits:
  - Fast to evaluate
  - Good convergence properties
  - Potentially local
- Problem:
  - Continuity between pieces

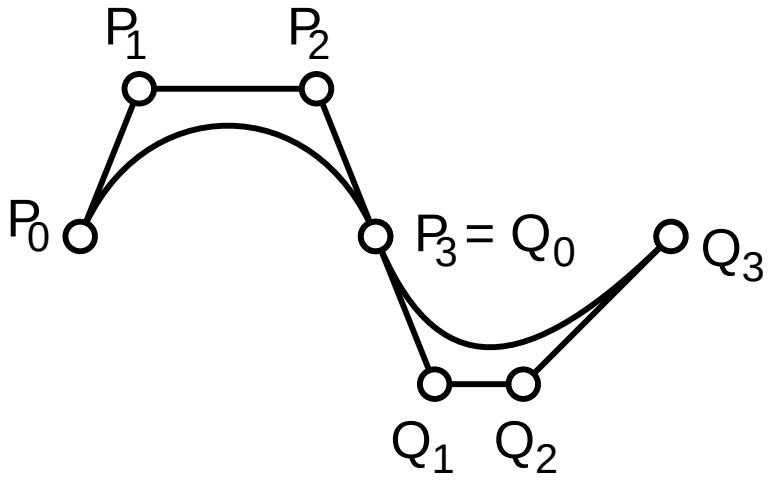
C^0 Piecewise Cubic Bézier

- Piecewise cubic
- Bézier form
- Bézier Property: curve interpolates first control point
- Bézier Property: curve interpolates last control point
- C^0: Set last control point of one segment to be first control point of next segment



C^1 Piecewise Cubic Bézier

- Need parameterizations of intervals to discuss C^1 continuity
- Assume uniform parameterization
  $[0,1]$, $[1,2]$, $[2,3]$, ...
- Need C^0 to have C^1
  $P_3 = Q_0$
- C^1 is $P_3 - P_2 = Q_1 - Q_0$



Cubic Hermite Interpolation

- Problem: Given points $P_0, \ldots, P_N$ and vectors $\vec{v}_0, \ldots, \vec{v}_N$
  Find: Piecewise C^1 cubic P s.t.

$$\begin{aligned} P(i) &= P_i \\ P'(i) &= \vec{v}_i \end{aligned}$$

  for $i = 0, \ldots, N$
- Solution: Use one cubic Bézier segment per interval
  Then

$$\begin{aligned} P_{i,0} &= P_i \\ P_{i,1} &= P_i + \frac{\vec{v}_i}{3} \\ P_{i,2} &= P_{i+1} - \frac{\vec{v}_{i+1}}{3} \\ P_{i,3} &= P_{i+1} \end{aligned}$$

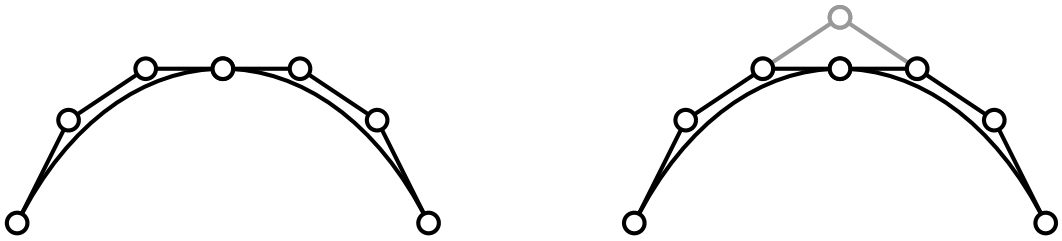
Catmull-Rom Splines

- When modelling, specifying derivatives can be either good or bad, but more commonly bad
- Want to specify just points
- Idea: make up derivatives from points, and use Cubic Hermite

- For $i = 1, \ldots, N-1$ let
$$\vec{v}_i = P_{i+1} - P_{i-1}$$
- Need derivatives for $i = 0, N$.
  - Discard data
  - Set to 0
  - Set to $P_1 - P_0$ (perhaps scaled)

C^2 Piecewise Cubic Bézier

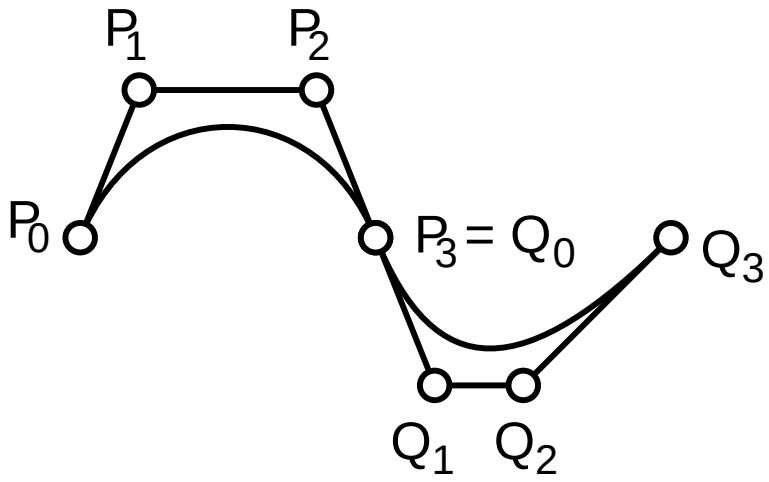
- With cubics, we can have C^2 continuity (why not C^3?)
- A-frame construction gives C^2 constraints on Bézier segments



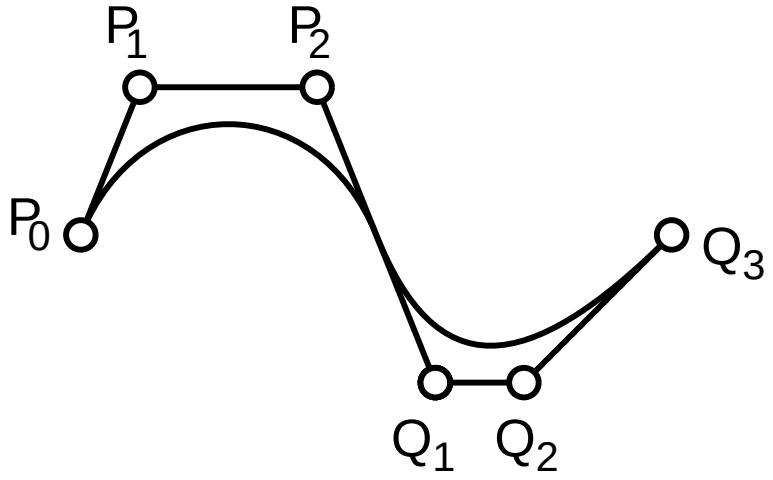
- Too hard to make user place points in this manner, so...
- Modelling: User places four control points, program places next three, user places one more, program places next three, ...
- Could hide program placed control points
- Redundant representation

C^1 Bézier Spline

- Suppose we have a C^1 Bézier spline

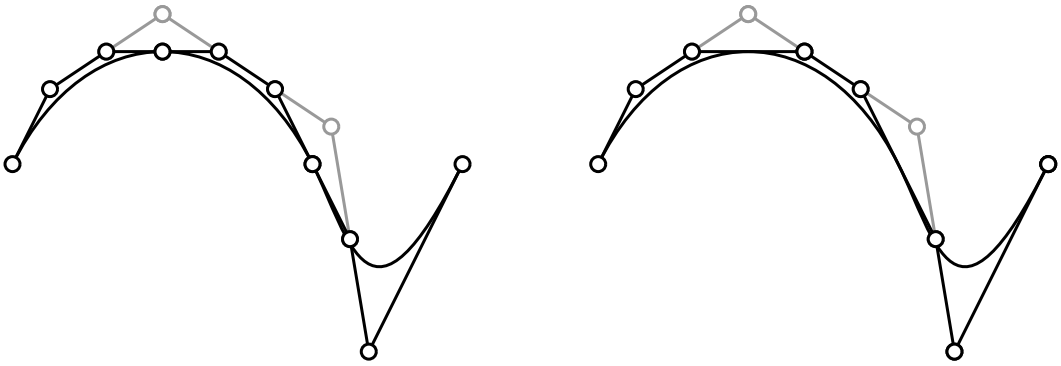


- For C^1, cubic Bézier spline, need only two control points per segment, plus one more at start and one more at end.

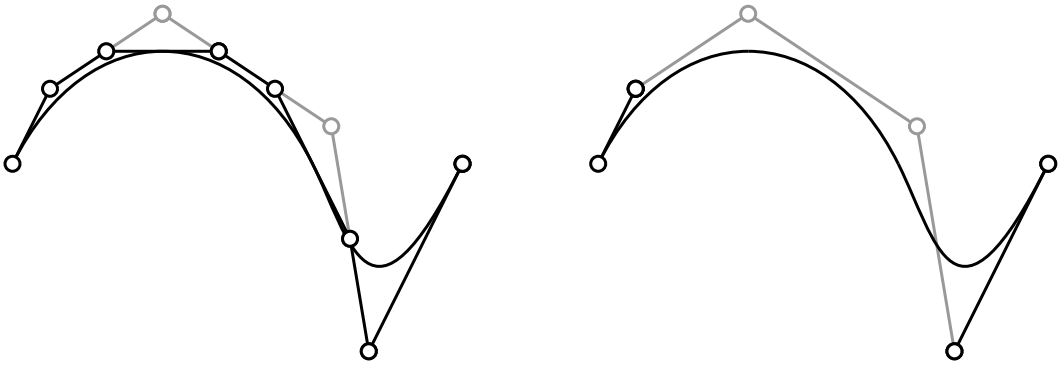


C^2 Bézier Spline

- Suppose we have a C^2 Bézier spline
  Joints can be computed as described for C^1

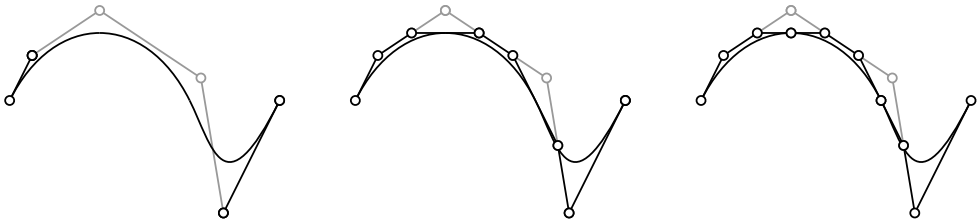


  If we keep the gray points, we can compute the black points



- Thus, for C^2, cubic Bézier spline, need only one control point per segment, plus two more at start and end.

- From gray points, we can recover Bézier control points



- The gray points are **B-spline** control points.

Evaluating B-splines

- Cubic B-splines give us C^2 continuity for little effort.
- To evaluate a B-spline, we could convert to Bézier and evaluate the Bézier curves.
- There is also a de Casteljau style evaluation
  The de Boor algorithm.
- Thus far, have considered the case for intervals $[0,1],[0,2],\ldots$
  (i.e., integers)
  One polynomial over each interval
- In general, B-splines can go over arbitrary intervals $[t_0,t_1],[t_1,t_2],\ldots$
  where $t_0 < t_1 < t_2 < \ldots$
  t_i are called *knots*
- B-splines have different basis functions

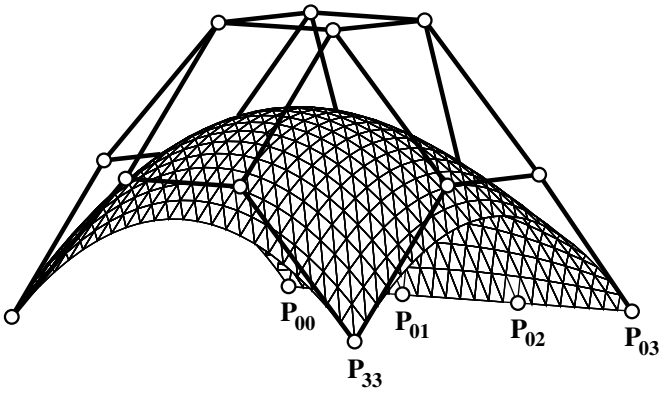
27.5 Tensor Product Patches

- **Control polygon** is polygonal mesh with vertices $P_{i,j}$
- **Patch basis functions** are products of curve basis functions

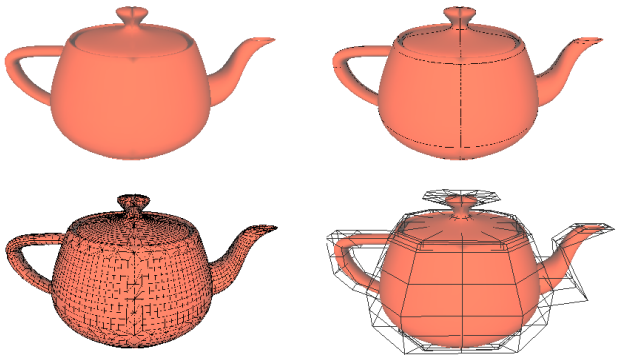
$$P(s,t) = \sum_{i=0}^{n} \sum_{j=0}^{n} P_{i,j} B_{i,j}^{n}(s,t)$$

where

$$B_{i,j}^{n}(s,t) = B_i^n(s) B_j^n(t)$$



The Teapot



Properties:

- Patch basis functions **sum to one**

$$\sum_{i=0}^{n}\sum_{j=0}^{n} B_i^n(s)B_j^n(t) = 1$$

- Patch basis functions are **nonnegative** on $[0,1] \times [0,1]$

$$B_i^n(s)B_j^n(t) \geq 0 \quad \text{for} \quad 0 \leq s,t \leq 1$$

$\Longrightarrow$ Surface patch is in the **convex hull** of the control points

$\Longrightarrow$ Surface patch is **affinely invariant**
(Transform the patch by transforming the control points)

Subdivision, Recursion, Evaluation:

- As for curves in each variable separately and independently
- **Tangent plane is not produced!**
  - Normals must be computed from partial derivatives.

Partial Derivatives:

- Ordinary derivative in each variable separately:

$$\frac{\partial}{\partial s}P(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{n} P_{i,j}\left[\frac{d}{ds}B_i^n(s)\right]B_j^n(t)$$

$$\frac{\partial}{\partial t}P(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{n} P_{i,j}B_i^n(s)\left[\frac{d}{dt}B_j^n(t)\right]$$

- Each is a **tangent vector** in a parametric direction

- The (unnormalized) **surface normal** is given at any regular point by

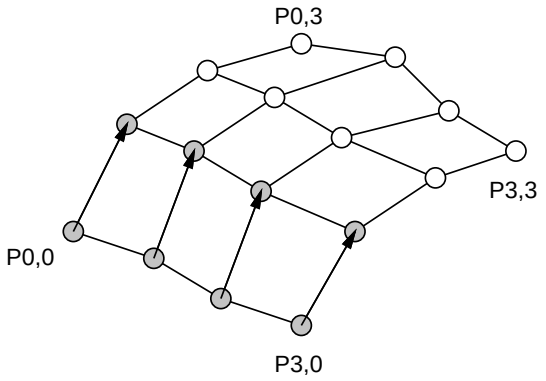
$$\pm \left[\frac{\partial}{\partial s} P(s,t) \times \frac{\partial}{\partial t} P(s,t) \right]$$

  (the sign dictates what is the **outward pointing normal**)

- In particular, the **cross-boundary tangent** is given by (e.g. for the $s = 0$ boundary):

$$n \sum_{i=0}^{n} (P_{i,1} - P_{i,0})\, B_i^n(t)$$

  (and similarly for the other boundaries)

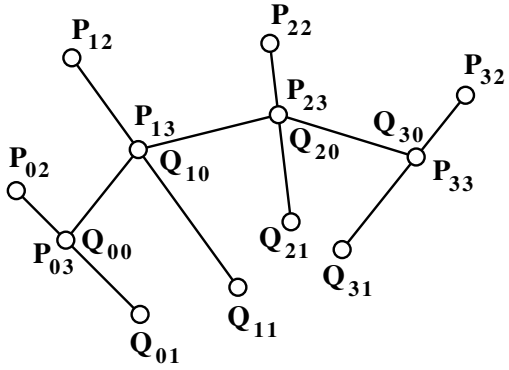


Smoothly Joined Patches:

- Can be achieved by ensuring that

$$(P_{i,n} - P_{i,n-1}) = \beta(Q_{i,1} - Q_{i,0}) \quad \text{for} \quad \beta > 0$$

  (and correspondingly for the other boundaries)



Rendering:

- Divide up into polygons:

A. By stepping

$$\begin{aligned} s &= 0, \delta, 2\delta, \ldots, 1 \\ t &= 0, \gamma, 2\gamma, \ldots, 1 \end{aligned}$$

and joining up sides and diagonals to produce a triangular mesh

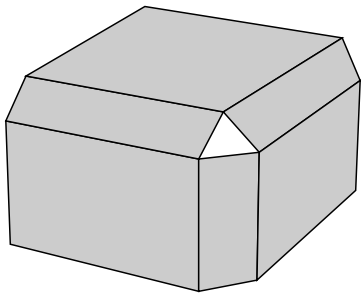
B. By subdividing and rendering the control polygon

Tensor Product B-splines

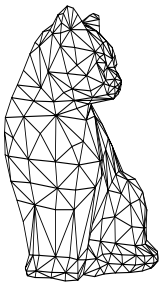
- Could use B-splines instead of Bézier
  Automatic continuity between patches

Problem with tensor product patches

- Work well for rectilinear data
- Problems arise when filling non-rectangular holes
  Suitcase Corner



- A lot of data is non-rectangular
  Triangular



27.6 Barycentric Coordinates

Coordinate Frames:

- Vector oriented; Derived from linear space basis
- One point and n vectors in space of dimension n: $D_n, \vec{v}_0, \ldots, \vec{v}_{n-1}$
  - Vectors $\vec{v}_i$ are *linearly independent*
  - Coordinates are weights of vectors, with weight of 1 for point

Barycentric Frames

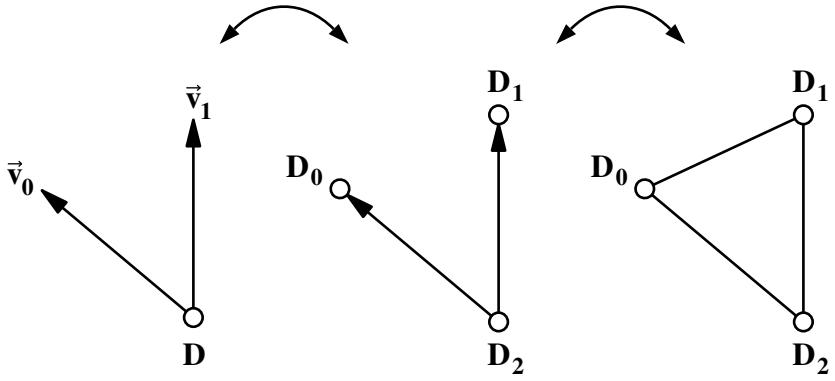
- Point oriented
- *n+1* points in space of dimension *n*: $D_0, \ldots, D_n$
  - Points are *in general position*
  - Coordinates are weights of points, with coordinates summing to 1

Frames of Both Types Are Equivalent

- Express each $\vec{v}_i$ as $D_i - D_n$ for $D_i = D_n + v_i$

$$\begin{aligned} P &= D_n + \sum_{i=0}^{n-1} p_i \vec{v}_i \\ &= D_n + \sum_{i=0}^{n-1} p_i (D_i - D_n) \\ &= (1 - \sum_{i=0}^{n-1} p_i) D_n + \sum_{i=0}^{n-1} p_i D_i \\ &= \sum_{i=0}^{n} w_i D_i \quad \text{where} \quad \sum_{i=0}^{n} w_i = 1 \end{aligned}$$

- And, of course, conversely



27.7 Triangular Patches

deCasteljau Revisited Barycentrically:

- Linear blend expressed in barycentric terms

$$(1-t)P_0 + tP_1 = rP_0 + tP_1 \quad \text{where} \quad r+t=1$$

- Higher powers:

$$P(t) \;=\; \sum_{i=0}^{n} P_i \left(\frac{n!}{i!(n-i)!}\right)(1-t)^{n-i}t^i$$

Symmetric Form of Bernstein Polynomials

$$\begin{aligned} P((r,t)) \quad &= \quad \sum_{\substack{i+j=n \\ i\geq 0, j\geq 0}} P_i \left(\frac{n!}{i!j!}\right) t^i r^j \quad \text{where} \quad r+t=1 \\ \Longrightarrow \quad &\sum_{\substack{i+j=n \\ i\geq 0, j\geq 0}} P_{ij} B^n_{ij}(r,t) \end{aligned}$$

Examples

$$\begin{aligned} \{B^0_{00}(r,t)\} &= \{1\} \\ \{B^1_{01}(r,t), B^1_{10}(r,t)\} &= \{r,t\} \\ \{B^2_{02}(r,t), B^2_{11}(r,t), B^2_{20}(r,t)\} &= \{r^2, 2rt, t^2\} \\ \{B^3_{03}(r,t), B^3_{12}(r,t), B^3_{21}(r,t), B^3_{30}(r,t)\} &= \{r^3, 3r^2t, 3rt^2, t^3\} \end{aligned}$$

Surfaces – Barycentric Blends on Triangles:

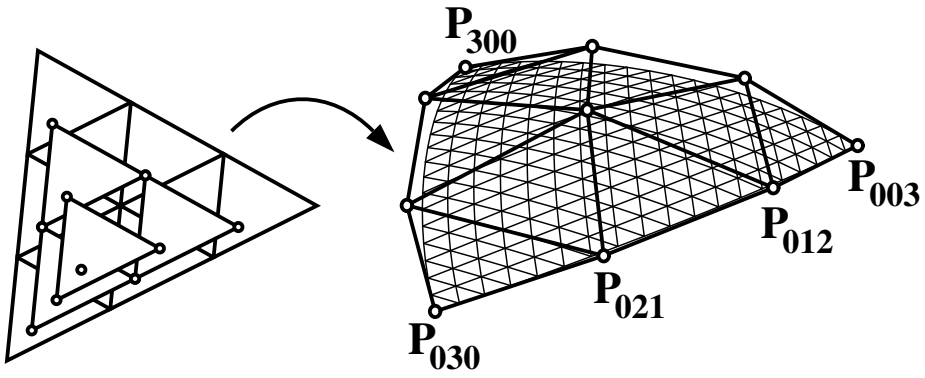
- Formulas:

$$\begin{aligned} P(r,s,t) \;&=\; \sum_{\substack{i+j+k=n \\ i\geq 0, j\geq 0, k\geq 0}} P_{ijk} B^n_{ijk}(r,s,t) \\ B^n_{ijk}(r,s,t) \;&=\; \frac{n!}{i!j!k!} r^i s^j t^k \end{aligned}$$

Triangular deCasteljau:

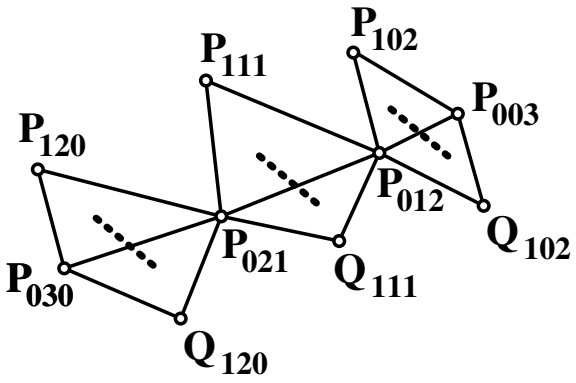
- Join adjacently indexed P_{ijk} by triangles
- Find $r:s:t$ barycentric point in each triangle

- Join adjacent points by triangles
- Repeat
  - Final point is the surface point $P(r, s, t)$
  - Final triangle is tangent to the surface at $P(r, s, t)$



Properties:

- Each boundary curve is a Bézier curve
- Patches will be joined smoothly if pairs of boundary triangles are affine images of domain triangles



Problems:

- Continuity between two patches is easy, but joining n patches at a corner is hard
- In general, joining things together with either tensor product or triangular patches is hard

27.8 Subdivision Surfaces

Introduction

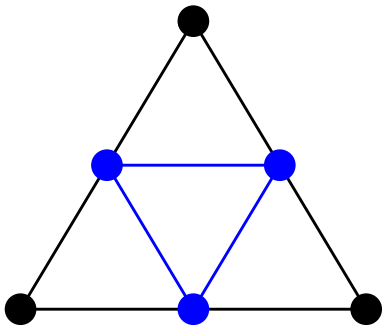
- Hard to piece spline surfaces together
- Hard to fit splines to arbitrary mesh
  n-sides faces, n-vertex neighbours
- Subdivision is an alternative surfacing scheme
  Fits surface to arbitrary mesh
  Only a few "gotcha's"
- Will look at two simple subdivision schemes
  Both schemes are simple to implement
  Neither scheme is one of "choice"
- (This material is from Chapter 7 of Warren, Weimer, Subdivision Methods for Geometric Design)

Polyhedron

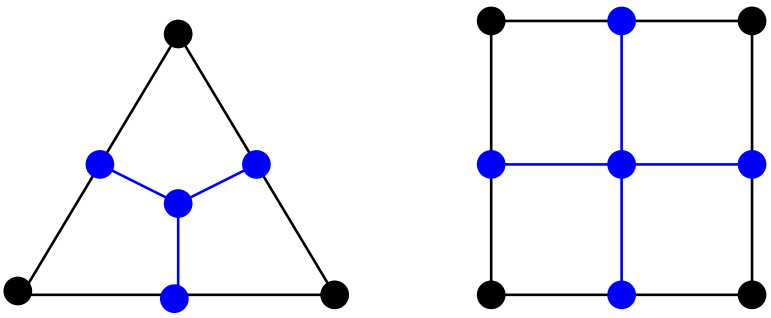
- Start with a description of a polyhedron
  List of vertices, list of faces
- Example: A cube
  Vertices:
  {{0,0,0}, {1,0,0}, {1,1,0}, {0,1,0}, {0,0,1}, {1,0,1}, {1,1,1}, {0,1,1}}
  Faces:
  {{1,4,3,2}, {1,2,6,5}, {2,3,7,6}, {3,4,8,7}, {4,1,5,8}, {5,6,7,8}}
- Valence(v): number of faces containing v
- Triangle mesh: all faces are triangles
- Quad mesh: all faces are quads

Topological Subdivision

- First step in subdivision is to split each face into new set of faces
  Will consider two types of splits (others exist)
- Triangular (linear) splits



- Quad (bilinear) subdivision

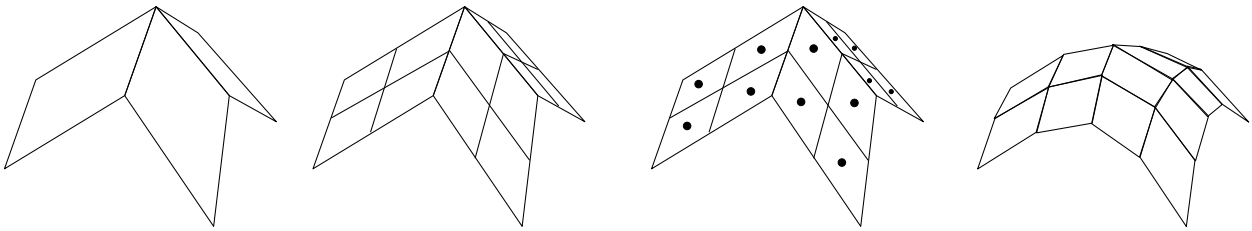


Bilinear Subdivision Plus Averaging

1. Perform bilinear subdivision of mesh
2. Compute centroid c_f of each quad face f
3. For each vertex v of subdivided mesh, set v to

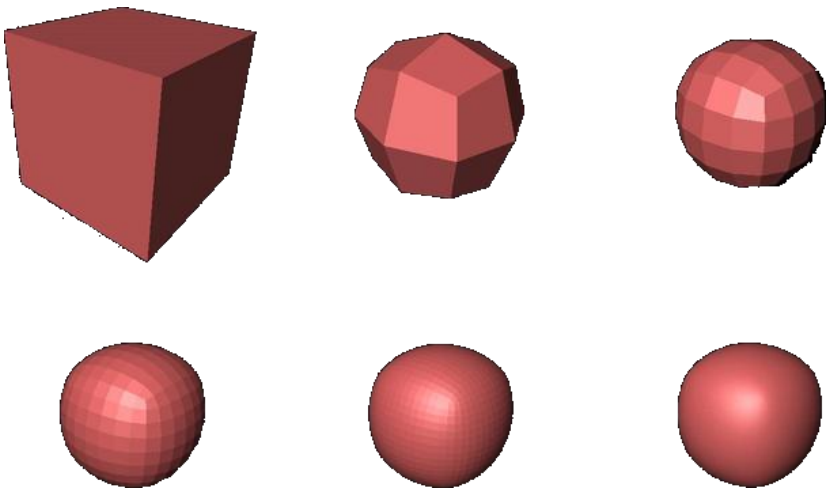
$$\frac{\sum_{f \in n(v)} c_f}{\#n(v)}$$

where $n(v)$ is the set of faces neighbouring v and $\#n(v)$ is the number of faces neighbouring v



(different rules used for mesh boundaries)

Catmull-Clark Example



- Start with a cube
- Repeatedly subdivide
- Do "something" to get normals

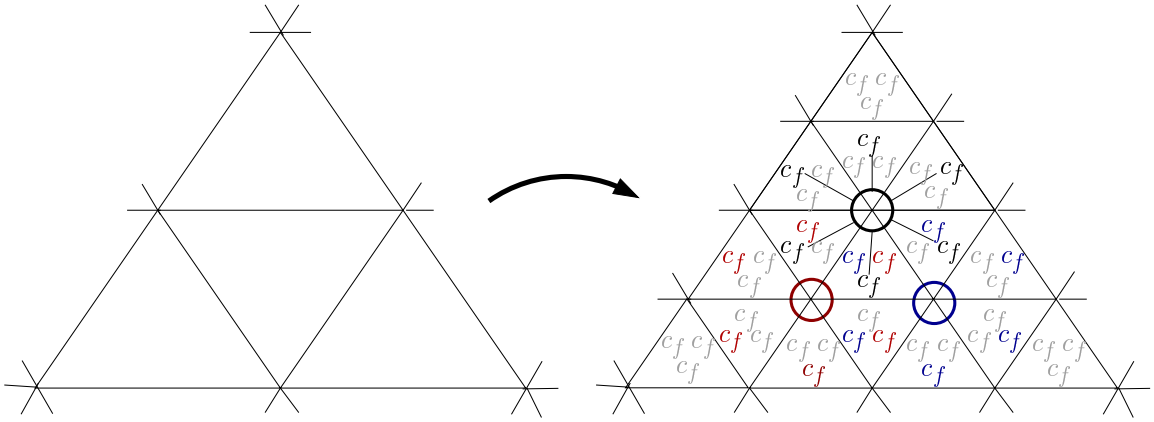
Linear Subdivision Plus Triangle Averaging

1. Perform linear subdivision of each triangle in mesh
2. For each vertex v of subdivided mesh,
   (a) For each triangle f neighbouring v compute

$$c_f = v/4 + 3v^+/8 + 3v^-/8$$

   where v^+ and v^- are the other two vertices of f
   (b) Set v to

$$\frac{\sum_{f \in n(v)} c_f}{\#n(v)}$$



Details, Issues

- The two schemes shown here are simple, but give poor surfaces
  Catmull-Clark and Loop are examples of better schemes
- Boundaries are handled using different rules
- Special rules to add creases to surface
- Major implementation issue is data structure
- Extraordinary vertices pose difficulties
  Valence other than 4 in quad mesh
  Valence other than 6 in tri mesh
- Mathematics behind the schemes is far more elaborate than the schemes themselves
- Parameterizations needed for texture mapping
- Normals?

27.9 Wavelets

Wavelets are a discrete form of Fourier series

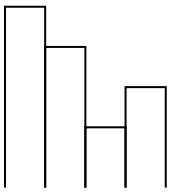
- Take a discrete signal (image).
- Decompose it into a sequence of frequencies.
- Use compact support basis functions rather than infinite support sin and cos.
- Construct a vector space using "unit pixels" as basis elements (piecewise constant functions).
- Average to get low frequencies.
- Construct "detail functions" to recover detail.
- Unit pixel bases form nested sequence of vector spaces, with the detail functions (wavelets) being the difference between these spaces.

1-D Haar

- Suppose we have the coefficients

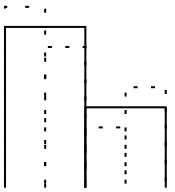
  $$[9\ 7\ 3\ 5]$$

  where we think of these coefficients as 1-D pixel values

  The simplest wavelet transform averages adjacent pixels, leaving

  $$[8\ 4]$$

- Clearly, we have lost information.

  Note that 9 and 7 are 8 plus or minus 1, and 3 and 5 are 4 minus or plus one.

  These are our detail coefficients:

  $$[1\ -1]$$

- We can repeat this process, giving us the sequence

Resolution	Averages	Details
4	[9 7 3 5]	
2	[8 4]	[1 -1]
1	[6]	[2]

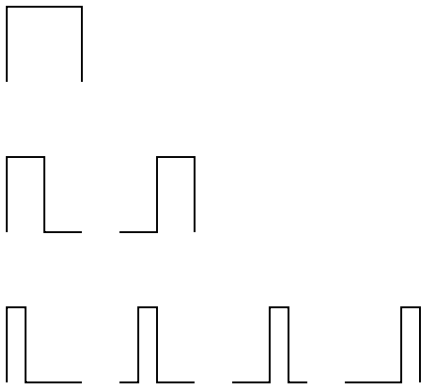
- The Wavelet Transform (wavelet decomposition) maps from [9 7 3 5] to [6 2 1 –1] (i.e., the final scale and all the details).
  - The process is called a Filter Bank.
  - No data is gained or loss; we just have a different representation.
  - In general, we expect many detail coefficients to be small. Truncating these to 0 gives us a lossy compression technique.

- Basis function for wavelets are called scaling functions.

  For Haar, can use

  $$\phi_i^j(x) = \phi(2^j x - i)$$

  where $\phi(x) = 1$ for $0 \leq x < 1$ and 0 otherwise.



  Support of a function refers to the region over which the function is non-zero.

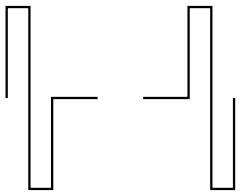
  Note that the above basis functions have *compact support*, meaning they are non-zero over a finite region.

- The Haar Wavelets are

  $$\psi_i^j(x) = \psi(2^j - i)$$

  where

  $$\psi(x) = \begin{cases} 1 & \text{for } 0 \leq x < 1/2 \\ -1 & \text{for } 1/2 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$



- Example again.
  Originally, w.r.t. V^2,

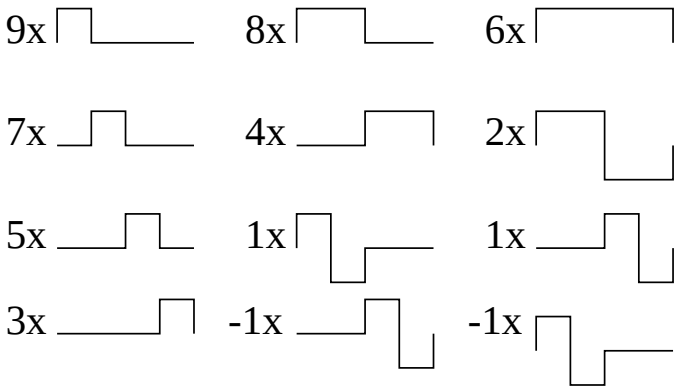
$$I(x) = 9\phi_0^2(x) + 7\phi_1^2(x) + 3\phi_2^2(x) + 5\phi_3^2(x)$$

  Rewriting w.r.t. V^1 and W^1,

$$I(x) = 8\phi_0^1(x) + 4\phi_1^1(x) + 1\psi_0^1(x) + (-1)\psi_1^1(x)$$

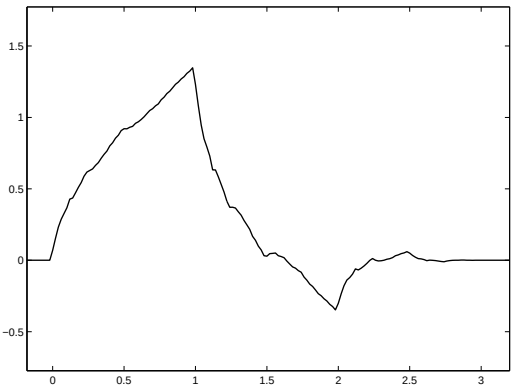
  Rewriting the ϕs in terms of V^0, W^0,

$$I(x) = 6\phi_0^0(x) + 2\psi_0^0(x) + 1\psi_0^1(x) + (-1)\psi_1^1(x)$$



Non-Haar Wavelets

- Haar is simple but doesn't have certain properties
- Non-Haar wavelets possible
- Example: Daubechies wavelet starts from



  The function is not as smooth as it looks!

- Haar often used in practice because it's simple, cheap

Wavelet Compression

- Lossless image compression – how do we represent an image using as few bits as possible?
  - Pigeon hole principle tells us we've lost before we start.
    Have to have expectations about image before you can achieve any compression.
  - Lossy compression involves two main steps: Losing part of the data, and then performing lossless compression on what's left.
- Can use wavelets to perform simple lossy part of compression.
  If we order coefficients in decreasing magnitude, then error is minimized.

2D Haar and Image Compression

- Standard decomposition: Apply 1-D Haar to each row. Then apply 1-D Haar to each column.
- Nonstandard decomposition: Apply one step of 1-D Haar to each row. Then apply one step of 1-D Haar to each column. Repeat on quadrant containing averages in both directions.
- Image compression in similar fashion, except it's expensive to sort coefficients.

28 Non-Photorealistic Rendering

28.1 2D NPR

- Traditional goal of computer graphics is photorealism
- Accurate depiction of reality isn't always the real goal
- How can computer graphics help us visualize, communicate, create art?
- We'll look at some applications of computer graphics in the creation of art

Painterly rendering

- Idea: create a painterly interpretation of a photograph
- Starting with a source image, apply a model of strokes to approximate it
- Possible goals:
  - Closeness of approximation
  - Use as little paint (number of strokes, total stroke length) as possible
  - Coverage of canvas

Paint by Relaxation

Could try to optimize directly, but it's easier to have a kind of greedy approach, painting a sequence of strokes

4 photographer images

Paint by numbers

Haeberli, SIGGRAPH 1990

- Interactive stroke placement (random displacement from pointer position)
- Stroke colours sampled from source image

3 face images

`http://thinks.com/java/impressionist/`

Extensions

Stroke orientation

Extensions

Painterly rendering with curved brushstrokes of multiple sizes

Hertzmann, SIGGRAPH 1998

- Create painting in layers: coarse to fine strokes (fill in detail where it's needed)
- Pick a set of stroke locations; draw strokes where painting is unlike source image
- Long, curved strokes flow along contours of constant colour (perpendicular to image gradient)
- Large set of intuitive customizable parameters to create painting styles (e.g., max stroke length)

Fast paint texture

Hertzmann, NPAR 2002

- Can also render a secondary bump map by drawing stroke texture instead of solid colour
- Bump map colour image using stroke texture image

4 stroke images
4 palm tree images
6 skeleton images

Stylization and abstraction of photographs

Santella and DeCarlo, SIGGRAPH 2002

- Stroke models follow image features, but they don't understand the image. How can we apply an NPR filter in a meaningful way?
- Let a person determine importance by having them look at the source image and tracking their eye movements
- Not strokes, but large regions of constant colour
- Edges drawn as well

Woman with guitar photo

Woman with guitar NPR image

Simulating natural media

- Implement a physical simulation to approximate the behaviour and appearance of a traditional artistic medium

Computer-Generated Watercolor

Curtis, Anderson, Seims, Fleischer and Salesin, SIGGRAPH 1996

- Very realistic simulation of the appearance of watercolour paints
- Treat the painting as a big fluid flow simulation
- Navier-Stokes equations to push fluid around on top of the paper
- Fluid and paper exchange pigment
- Paper propagates water via capillary action

Photo of watercolour brush strokes

Real paint

Simulation of watercolour brush strokes

Simulated

Samples of watercolour paint simulations

Fruit, and lots of it!

Pencil Rendering

Sousa and Buchanan, GI 1999, CGF 2000

- Simulation of pencil lead and paper, derived from close observation of real materials
- Model the geometry of the pencil tip, pressure on page
- Graphite particles can move to and from paper, so system can model pencils, erasers, and blenders
- Pressure affects geometry of paper, and therefore appearance of future strokes

Real and simulated pencil

Crumpled paper

photo and 2 NPR pencil sketches of girl

Batik

Moose

Pen and Ink Illustration

- Problem: each stroke must simultaneously convey texture and tone

Stroke textures

Example textures

Wall images

Approximately constant t
space

View-dependent rendering of surface texture

Wall images

Wall images

Lighting-dependent rendering for shadows and accenting

House

Two more houses

Indication

Interactive pen-and-ink

Examples

Examples

Examples

Examples Examples

Orientable textures

Paint bushes

Racoon

28.2 3D NPR

- Support real-time non-photorealistic rendering of 3D scenes via hardware acceleration

HijackGL and NPRQuake

NPR Quake NPR Quake

NPR Quake

Real-time hatching

Six examples

Image-space silhouettes

- Silhouettes are discontinuities in the depth image
- Creases are second order discontinuities in the depth image
- The right image on which to do edge detection

Hex nut exampl

Funky box examples

- Creases can also be seen as discontinuities in the normal image

Bust

Object-space silhouettes

- Given a polyhedral mesh and a camera position, find and draw all visible silhouettes
- Two subproblems:
  - Find all possible silhouettes relative to the current viewpoint
  - Process these silhouettes to determine visibility

Silhouettes

- What is a silhouette? For a smooth object, it's a point where the vector from the camera just grazes the point.
- In other words:

$$(p - c) \cdot n = 0$$

  Where p is the point on the surface,
  c is the location of the camera,
  and n is the normal at point p.

Silhouettes on meshes

- A mesh with face normals is not smooth
- Approximate silhouettes by finding mesh edges that separate a front face from a back face
- Slow to check all edges every frame, so use some kind of acceleration (edge buffer, probabilistic checking, ...)
- Can also identify silhouettes on smooth mesh (one with vertex normals)

Silhouette visibility

- Just because an edge is a silhouette doesn't mean it's visible:
  - Occlusion by another object
  - Occlusion by another part of the same object
  - Partially visible
- Need to decide, for every silhouette edge, which parts are visible

Image-space visibility processing

- Turn on depth buffering
- Draw all mesh faces in background colour
- Draw all silhouette edges, give every edge a unique colour that identifies it
- Frame buffer now holds pixel-accurate image of which pieces of silhouettes are visible

Stylized silhouettes

- Use visibility to guide placement of stylized strokes along paths
- Parameterization and temporal coherence are still problems

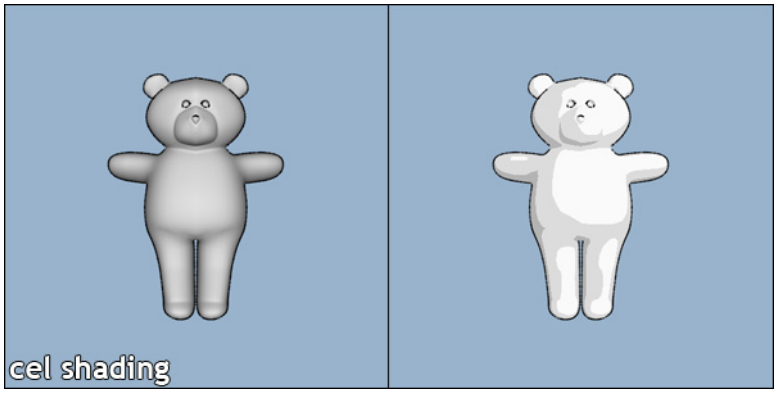
Toon shading

- Cartoons typically use a small range of colours for each object in a scene
  - Shadow, body, and sometimes highlight
- Easy to simulate in ray tracer or real-time
  - Quantize by diffuse attenuation to two or three colours

Elf girl Roses

Toon shading in OpenGL

- Perform diffuse lighting computation at every vertex in software
- Use diffuse brightness as parameter to a one-dimensional texture map
  - Map has only two or three colours
- Sample the texture with nearest-neighbour interpolation: deliberately not blended
- Could definitely be implemented directly on the GPU as a fragment program



Recent work

- WYSIWYG NPR (SIGGRAPH 2002)
- Coherent Stylized Silhouettes (SIGGRAPH 2003)
- Suggestive Contours (SIGGRAPH 2003)

29 Animation

29.1 Overview

Computer Animation

Animation: Rapid display of slightly different images create the illusion of motion.

Conventional approach: Keyframed animation

Keyframes, inbetween frames

Computer Assisted:

- Human key frames, computer inbetweening
- Supporting techniques:
  Squish-box deformations and skeletons, motion capture, kinematics and inverse kinematics.
- Other animation approaches:
  Physically-based dynamics, constraint-based animation, procedural animation, behavioral animation.

29.2 Traditional 2D Cel Animation

Traditional 2D Cel Animation

- Traditional animation is labor intensive.
  - Start with story board.
  - Master artist draws keyframes to define action.
  - Sweat-shop artists draw inbetween frames to complete animation.
- A few traditional animation rules:
  - Stretch and squash (anti-aliasing), timing, secondary actions
  - Exaggeration, appeal, follow through
  - Anticipation, staging.
  - Be careful with camera
    - Smooth changes
    - Few changes
- Disney was the most successful, but it was high risk.

29.3 Automated Keyframing

- Replace the human inbetweener with interpolation algorithms.
- Keyframes correspond to settings of parameters at different points in time.

\+ The computer provides repeatability and automatic management of keyframes.

– Interpolation is not as smart as a human inbetweener.

Animator may have to provide more key frames and/or additional information to get a good result.

Utility

A good keyframe system limits the number and maximizes the utility of key parameters.

- Parameters should have immediate geometric or visible significance.

  Entries in a transformation matrix are *not* good parameters.
- Static properties maintained automatically.
- Relationships between parts of articulated objects maintained automatically.
- Interpolation routines geared towards parameter types.

  Motion paths, orientation, camera attitude, and surface properties: different types of control.
- Real-time interface to interactively set parameters and iteratively improve the animation.

29.4 Functional Animation

- Independent scalar function specified for each "keyframed" parameter.
- Functional animation is a basic capability that supports all others.
- Most useful for truly scalar quantities: brightness of a light source, for example.
- Splines are useful for representing functions.
- Continuity control is a must: both its automatic maintenance and selective breaking.
- The functions can be edited both *explicitly*, as graphs, or *implicitly*, through a direct manipulation interface.

Linear Interpolation

- Basic interpolation technique for supporting animation is linear interpolation, or the *lerp*.
- Given two parameters p_0 and p_1 at times t_0 and t_1, an intermediate value is

$$p(t) = \frac{t_1 - t}{t_1 - t_0} p_0 + \frac{t - t_0}{t_1 - t_0} p_1.$$

- Advantages and Disadvantages:

  – Discontinuities in derivative exist at all key frame points.

  \+ The rate of change within a segment is constant (and so can easily be controlled).

Controlling Velocity

Given a constant rate of change, can create *variable* rate of change.
Given any function $\tau = f(t)$, reparameterize with τ.
For the lerp,

$$p(\tau) = p(f(t)) = \frac{f(t_1) - f(t)}{f(t_1) - f(t_0)} p_0 + \frac{f(t) - f(t_0)}{f(t_1) - f(t_0)} p_1.$$

Spline Interpolation

Instead of linear interpolation, spline interpolation can be used.

\+ Continuity control can be obtained.

\+ Fewer keyframes may be required for a given level of "quality".

– Control points for the splines may have to be computed based on interpolation constraints at the control points.

– Extra information may be required at keyframes, such as tangent vectors.

– Splines are more expensive to evaluate.

– Splines are more difficult to implement.

Spline Types for Animation

Different spline types will have different tradeoffs and capabilities:

B-splines: Give automatic continuity control, but only approximate their control points.

Non-uniform B-splines: Are needed to break continuity.

Non-uniform Rational B-splines (NURBS): Provide additional local control over the shape of a curve.

Hermite splines: Require tangent vectors at each keyframe.

Catmull-Rom splines: Provide only first derivative continuity.

Transformation Matrix Animation

- One way to support an animation capability: interpolate a transformation matrix.
- Keyframes would be "poses" of objects given by the animator.
- Functional animation would be applied independently to all entries of the transformation matrix.

Unfortunately, does not support animation of rigid bodies.

- Given two poses of an object that differ by a 180° rotation.
- Under linear interpolation, object will not rotate but will turn inside out
  collapses to a plane in the middle of the interpolation.

Rigid Body Animation

- Rigid body transformations only have 6 degrees of freedom
  General affine transformations have 12.
- To obtain good interpolation we have to consider separately
  - three degrees of freedom from translation
  - three that result from orientation.

29.5 Motion Path Animation

We would like to translate a point through space along a given path. We would like:

- Independent control of velocity along path.
- Continuity control.

While splines can easily support continuity control, *velocity* control is more difficult.

Motion Path Spline Interpolation Problems

Spline position interpolation gives continuity control over changes position, but:

- Visually notice discontinuities in 2nd derivative – need C^3 splines (degree 4).
- Equal increment in spline parameter does not correspond to equal increment in distance along the spline.
- Different segments of the spline with the same parametric length can have different physical lengths.
- If we parameterize the spline directly with time objects will move at a non-uniform speed.

Arc Length Parameterization

1. Given spline path $P(u) = [x(u), y(u), z(u)]$, compute arclength of spline as a function of u: $s = A(u)$.
2. Find the inverse of $A(u)$: $u = A^{-1}(s)$.
3. Substitute $u = A^{-1}(s)$ into $P(u)$ to find a motion path parameterized by arclength, s: $P(s) = P(A^{-1}(s))$.

Note that u (and thus s) should be global parameters, extending across all segments of the original spline.

Velocity Control

To control velocity along a spline motion path,

- Let $s = f(t)$ specify distance along the spline as a function of time t.
- The function $f(t)$, being just a scalar value, can be supported with a functional animation technique (i.e., another spline).
- The function $f(t)$ may be specifed as the integral of yet another function, $v(t) = df(t)/dt$, the *velocity*.

  Integral of a spline function can be found analytically, through a manipulation of the control points.
- The motion path as a function of time t is thus given by $P(A^{-1}(f(t)))$.

Problems with Arc-Length Parameterization

1. The arc-length $s = A(u)$ is given by the integral

$$s = A(u) = \int_0^u \sqrt{\left(\frac{dx(v)}{dv}\right)^2 + \left(\frac{dy(v)}{dv}\right)^2 + \left(\frac{dz(v)}{dv}\right)^2}\, dv$$

   No analytic solution if the motion path is a cubic spline.

2. Since $A(u)$ has no analytic form, $A^{-1}(s)$ has no analytic form.

Issues in Real-Time Animation

- Exact arc-length parameterization not feasible.
- An alternative: compute points on the spline at equally-spaced parametric values, and use linear interpolation along these chords.
- The linear interpolation should consider the distance between samples to maintain constant velocity.

29.6 Orientation and Interpolation

Interpolate angles rather than transformation matrices.

- In two dimensions, only one angle is involved and animation is straightforward.
- In three dimensions, orientation requires three degrees of freedom.

  Interpolation is much nastier and harder to visualize.

Approaches

Two standard approaches to the orientation interpolation problem. Both related to problem of specifying orientation in the first place:

Euler angles. Three angles: x-roll followed by y-roll followed by z-roll.

- – Has defects: parameterization singularities, anisotropy, "gimbal lock", unpredictable interpolation.
- – Hard to solve inverse problem: given orientation, what are the angles?
- \+ Widely used in practice.
- \+ Easy to implement.
- \+ Inexpensive computationally.

Quaternions. Four-dimensional analogs of complex numbers.

- \+ Isotropic: avoid the problems of Euler angles.
- \+ Inverse problem easy to solve.
- \+ The user interface tends to be simpler.
- – More involved mathmatically.
- – Interpolation can be expensive in practice.

Euler Angles

- With $c_a = \cos(\theta_a)$ and $s_a = \sin(\theta_a)$,

$$\begin{aligned} R(\theta_x, \theta_y, \theta_z) &= \begin{bmatrix} c_y c_z & c_y s_z & -s_y & 0 \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y & 0 \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= R_x(\theta_x) R_y(\theta_y) R_z(\theta_z), \end{aligned}$$

  where $R_x(\theta_x)$, $R_y(\theta_y)$ and $R_z(\theta_z)$ are the standard rotation matrices.

- Given a point P represented as a homogeneous row vector, the rotation of P is given by $P' = PR(\theta_x, \theta_y, \theta_z)$.
- Animation between two rotations involves simply interpolating independently the three angles θ_x, θ_y, and θ_z.

Problems with the Euler angle approach include:

Parametric Singularity: A degree of freedom can suddenly vanish.

Anisotropy: The order of the axes is important.

Nonobviousness: The parameters lack useful geometric significance.

Inversion: Finding the Euler angles for a given orientation is difficult (not unique).

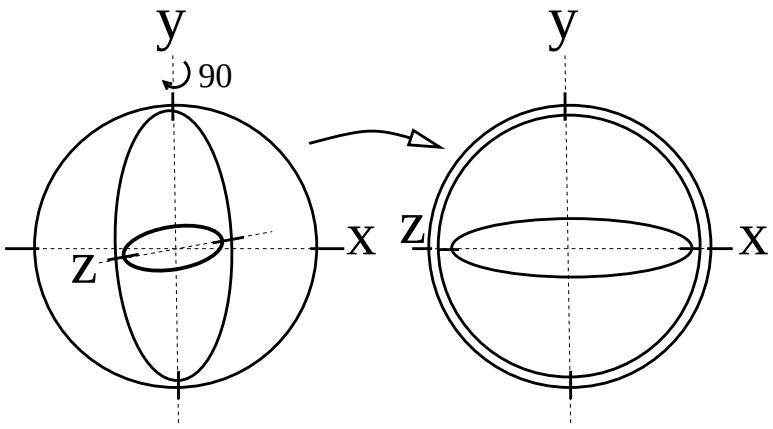
Coordinate system dependence: The orientation of the coordinate axes is important.

Gimbal Lock: An Example

- *Gimbal lock* is an example of a **parametric singularity**.
- Gimbal lock is a mechanical problem that arises in gyroscopes as well as quaternions.
- Set $\theta_y = \pi/2 = 90^\circ$, and set θ_x and θ_z arbitrarily. Then $c_y = 0$, $s_y = 1$ and the matrix $R(\theta_x, \pi/2, \theta_z)$ can be reduced to

$$\begin{aligned} R(\theta_x, \theta_y, \theta_z) &= \begin{bmatrix} 0 & 0 & -1 & 0 \\ s_x c_z - c_x s_z & s_x s_z + c_x c_z & 0 & 0 \\ c_x c_z + s_x s_z & c_x s_z - s_x c_z & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & -1 & 0 \\ \sin(\theta_x - \theta_z) & \cos(\theta_x - \theta_z) & 0 & 0 \\ \cos(\theta_x - \theta_z) & \sin(\theta_x - \theta_z) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \end{aligned}$$

- A y-roll by $\pi/2$ rotates the x-axis onto the negative z axis, and so a x-roll by θ has the same effect as a z-roll by $-\theta$.



- Gimbal lock can be very frustrating in practice:
  - During interactive manipulation the object will seem to "stick";
  - Certain orientations can be hard to obtain if approached from the wrong direction;
  - Interpolation through these parametric singularities will behave strangely.

29.7 Quaternions

- Four-dimensional analogs of complex numbers.
- Can be used to represent orientation without a directional selectivity.
- **Recall:** multiplication of complex numbers represents orientation and rotation in the plane.
- The rectangular and polar form of a complex number p are

$$p = a + bi = Ae^{i\theta}.$$

- Multiplication is a rotation about the origin:

$$p_1 p_2 = A_1 A_2 e^{i(\theta_1+\theta_2)}$$

Quaternions — Definitions

- Quaternions defined using three imaginary quantities: i, j, and k:

$$q = a + bi + cj + dk.$$

- Rules for combining these imaginary quatities:

$$\begin{aligned} i^2 = j^2 = k^2 &= -1, \\ ij = -ji &= k, \\ jk = -kj &= i, \\ ki = -ik &= j. \end{aligned}$$

- These rules are sufficient to define multiplication for quaternions.
- Quaternion multiplication *does not commute.*
- A quaternion can be broken into a scalar and a vector part:

$$q = (s, \vec{v}) = s + v_1 i + v_2 j + v_3 k.$$

- In vector notation, product of two quaternions is

$$q_1 q_2 = (s_1 s_2 - \vec{v}_1 \cdot \vec{v}_2, s_1 \vec{v}_2 + s_2 \vec{v}_1 + \vec{v}_1 \times \vec{v}_2).$$

- Quaternions also have conjugates: $\bar{q} = (s, -\vec{v})$.
- The norm of a quaternion is defined by

$$|(s, \vec{v})| = \sqrt{s^2 + v_1^2 + v_2^2 + v_3^2}.$$

Unit Quaternions

- Unit quaternions have unit norms
- Isomorphic to orientations
- The unit quaternion given by
$$q_r = (\cos(\theta), \sin(\theta)\vec{v})$$
is equivalent to a rotation by an angle of 2θ around the unit $\vec{v}$.
Note that q_r is equivalent to $-q_r$ when interpreted as an orientation.

Rotations and Quaternions

- Points in space can be represented by quaternions with a zero scalar part.
- Point P is represented by the quaternion
$$q_P = (0, P).$$
- The rotation of a point P by q_r is
$$q_{P'} = q_r q_P q_r^{-1}.$$
- For unit quaternions, the inverse is equivalent to the conjugate.

Advantages of Quaternions

- Doing rotation this way is fairly perverse, except for three points:
  1. For many applications, the independent definition of an axis of rotation and an angle makes sense.
  2. The definition of an orientation by a quaternion is coordinate system independent and isotropic.
  3. Spherical interpolation of quaternions gives better results than interpolation of Euler angles.

Interpolating Unit Quaternions

- Linear interpolation on all the entries of the quaternion does not have unit norm.
Angular velocity is not constant if we renormalize.
- Spherical linear interpolation (*slerp*):
  - Consider the quaternions as vectors, and find the angle between them:
$$\omega = \cos^{-1}(q_1 \cdot q_2).$$
  - Given a parameter $u \in [0, 1]$, the slerp is
$$q(u) = q_1 \frac{\sin((1-u)\omega)}{\sin(\omega)} + q_2 \frac{\sin(u\omega)}{\sin(\omega)}.$$

- Numerical difficulties when $\omega \approx 0$.
  In such as case, replace the slerp with a lerp.
- Also problems with $\omega \approx n\pi/2$;
  This should probably be flagged with an error and/or a request for more keyframes.

29.8 Animating Camera Motion

- Requirements for camera motion are different from object motion:
  - Specification of camera motion has a long cinematic tradition that should be respected.
  - The camera should always be level, unless we specify otherwise.
  - The image of objects of interest should be stable on the film plane.

Camera Cinematic Terminology

Dolly: Move forward, along the line of sight of the camera (towards the object of interest).

Track: Move horizontally, perpendicular to the line of sight of the camera. More generally, move in a horizontal plane.

Crane: Raise the camera vertically.

Tilt (Bank): Rotate about the horizontal axis perpendicular to the line of sight of the camera.

Pan (Yaw): Rotate about the vertical axis of the camera (after tilt).

In addition, cameras have other parameters that can be animated:

Zoom (in, out): Change the angular field of view of the camera.

Focus: Change the focal depth, i.e. the distance at which objects are in focus.

Depth of Field: Changing the aperture size has the effect of changing the depth of field, i.e. the range of depths over which objects can be considered to be in focus.

Camera Cinematics

- An animated zoom changes perspective, which can be disturbing.
  Use *dolly* to enlarge the image of an object of interest.
- The camera should almost never be rotated about its view direction
  Unless a seasick audience is the objective.
- Changing the focal depth can be used to track objects of interest,
  but a sophisticated renderer is required to simulate focus.
- Depth of field is rarely changed in real cinematography due to technical limitations,
  but might be useful in computer animation.

- Use smooth spline animation for camera animation.

  Quick camera moves should normally be replaced by cuts

  (instantaneous changes of scene) unless a special effect is desired.

- Animation of spotlight sources is similar to the animation of a camera.

  Support of a "light's-eye-view", therefore, is often useful.

Camera Animation Specification

- With a camera, usually more interested in what it's looking at than its exact orientation
- Camera animation may be specified by
  - A motion path for the camera itself.
  - A motion path for a lookat point
    (possibly derived from the motion of an object of interest).
- Camera moved along the motion path and the orientation of camera determined by the lookat point.
- Ideally, "focus would be pulled" to match the distance to object of interest.
- Tilt might be constrained to lie within certain bounds.

29.9 Tools for Shape Animation

Skeletons

- Given a model with large number of vertices, vertices can be grouped and manipulated as a unit.

  Example: in an arm, all points describing the forearm move more-or-less as a rigid body.

- Connectivity between groups can be represented by tying each group to a "bone" and arranging bones in an articulated "skeleton".
- The bone and all its associated vertices are treated as a single rigid object for the purposes of transformation.
- Movement of the bones is constrained by the joints in the skeleton.
- Different kinds of joints (revolute, hinge, ball) can support specific types of motion.
- An animation can be previewed using only the bones, speeding rendering time.
- In the final rendering, the model is shown without bones.
- Enhancement: vertices may be influenced by more than one bone.

  Results in more flexible and realistic surface near joints.

  Example: a kneecap and its skin surface affected by both the femur and shinbone, and assume a position halfway between either.

Free-Form Deformations

- Sometimes, skelton cannot capture desired shape change of an object.
- Example: The "sqaush and stretch" of a bouncing ball as it hits the ground.
- Such global changes in shape can be expressed using a *deformation.*
- A deformation
  - Changes the *space* around an object
  - Nonlinear transformation
  - New coordinates for every point in space are determined as functions of the old coordinates.

Process

1. Place rectangular "squish box" object to be animated.
2. The coordinates of all points within the box are determined relative to the frame given by the box.

   Suppose a vertex P can be expressed as $[u, v, w]$ relative to the coordinates of the box.
3. The new coordinates of P are given by a tensor product Bézier spline $B^{n_1,n_2,n_3}(u, v, w)$ with control points $P_{i,j,k}$.
4. If the $P_{i,j,k}$ have coordinates

   $$P_{i,j,k} = [i/(n_1 + 1), j/(n_2 + 1), k/(n_3 + 1)],$$

   the transformation is given by the identity:

   $$[u, v, w] = B^{n_1,n_2,n_3}(u, v, w).$$

   Normally the control points are initially set to these values and are moved to effect a deformation.

FFD Issues

- Continuity conditions can be enforced.

  Example: A hand is being deformed but the arm is not.

  The control points along the edge of the box that cuts the wrist should not be moved, nor the next layer.

  This maintains both position and derivative continuity across the wrist.
- The object should be finely tesselated or radical deformations will not work properly.
- Not volume preserving.

  For realistic animation, typically only small deformations are used.
- Modeling UI non-trivial

  Manipulating control points inadequate

Skeletons and FFDs

- Skeletons and free-form deformations can be used simultaneously.
- The number of control points in a free-form deformation can be large
- If moved in groups relative to skeleton, excellent animation control can be obtained for arbitrary models.

29.10 Kinematics and Inverse Kinematics

Kinematics and Inverse Kinematics

Kinematics: The study of motion independent of the forces that cause the motion. Includes position, velocity, acceleration.

Forward kinematics: The determination of the

- positions,
- velocities,
- accelerations

of all the links in an articulated model given the

- position,
- velocity,
- acceleration

of the root of the model and all the transformations between links.

- Forward kinematics is a necessity for skeletal keyframed animation
- Easy to implement.

Inverse kinematics: The derivation of the motion of intermediate links in an articulated body given the motion of some key links.

Inverse Kinematics

- Often nonlinear, underdetermined or overdetermined, possibly ill-conditioned.
- Complexity of determining solution proportional to number of free links.
- One free joint between two fixed ones can be solved fairly efficiently, for example, with only one spare degree of freedom.
- Extra constraints may be needed to obtain a unique and stable solution.
  - Example: Requiring a joint to point downwards as a gross approximation to gravity
  - Joint motion constraints (hinge vs. revolute vs. ball) are also useful.

- Additional optimization objectives
  - Resulting optimization problem solved iteratively as an animation proceeds.
  - Example optimization objectives:
    * minimize the kinetic energy of structure;
    * minimize the total elevation of the structure;
    * minimize the maximum (or average) angular torque.

29.11 Physically-Based Animation

Idea: to obtain a physically plausible animation, *simulate* the laws of Newtonian physics.
In contrast to kinematics, this is called a *dynamics* approach.

- Have to control objects by manipulating forces and torques, either directly or indirectly.
- Animated objects may be passive: bouncing balls, jello, wall of bricks falling down, etc.

Inverse dynamics is difficult

- Finding forces to move a physical object along a desired path
- Most current applications are "passive" variety
- Useful for secondary effects in keyframe animation:
  - a mouse character's ears should flap when it shakes its head,
  - a T. Rex's gut should sway while it runs,
  - a feather in a cap should wave when wind blows on it,
  - cloth should drape around an actor and respond to his/her/its movements,
  - a wall of bricks should fall down when a superhero crashes into it.

Caution: simulated secondary effects can make bad keyframed animation look even worse!

Simulation

Setting up and solving a physically-based animation proceeds as follows:

1. Create a model with physical attributes: mass, moment of inertia, elasticity, etc.
2. Derive differential equations by applying the laws of Newtonian physics.
3. Define initial conditions, i.e., initial velocities and positions.
4. Supply functions for external forces (possibly via keyframing).
5. Solve the differential equations to derive animation, i.e., motion of all objects in scene as a function of time.

- During solution of the differential equations we have to be prepared to deal with discontinuities due to collisions.
- We will discuss the simulation of spring-mass and point mass models.
- Both have useful applications and are relatively simple to simulate.

Spring-Mass Models

Create a model:

- Composed of a mesh of point masses m_i connected by springs with spring constants k_{ij} and rest lengths ℓ_{ij}.
- Let $P_i(t) = [x_i(t), y_i(t), z_i(t)]$ be the position of mass m_i at time t.
- Let N_i be all masses connected to mass m_i.

  If $j \in N_i$, then m_i and m_j are connected by a spring.
- Springs exert force proportional to displacement from rest length, in a direction that tends to restore the rest length:

$$\begin{aligned} \vec{f}^s_{ij}(t) &= -k_{ij}(\ell_{ij} - |P_i(t) - P_j(t)|)\frac{P_i(t) - P_j(t)}{|P_i(t) - P_j(t)|}, \\ \vec{f}^s_i(t) &= \sum_{j \in N_i} \vec{f}^s_{ij}(t). \end{aligned}$$

- Masses assumed embedded in a medium that provides a damping force of

$$\vec{f}_d = -\rho_i \vec{v}_i(t),$$

  $\vec{v}_i(t)$ is velocity of m_i at time t.
- Motion of each mass governed by second order ordinary differential equation:

$$m_i \vec{a}(t) = -\rho_i \vec{v}(t) + \vec{f}^s_i(t) + \vec{f}^e_i(t).$$

  $\vec{f}^e_i(t)$ is the sum of external forces on node i and

$$\begin{aligned} \vec{a} &= \left[\frac{d^2x_i(t)}{dt^2}, \frac{d^2y_i(t)}{dt^2}, \frac{d^2z_i(t)}{dt^2}\right], \\ \vec{v} &= \left[\frac{dx_i(t)}{dt}, \frac{dy_i(t)}{dt}, \frac{dz_i(t)}{dt}\right] \end{aligned}$$

- Initial conditions: user supplies initial positions of all masses and velocities.
- The user supplies external forces: gravity, collision forces, keyframed forces, wind, hydrodynamic resistance, etc. as a function of time.
- Simulation.
  - Factor second-order ODE into two coupled first-order ODE's:

$$\begin{aligned} \vec{v} &= [v_{xi}(t), v_{yi}(t), v_{zi}(t)] \\ &= \left[\frac{dx_i(t)}{dt}, \frac{dy_i(t)}{dt}, \frac{dz_i(t)}{dt}\right], \\ \vec{a} &= \left[\frac{dv_{xi}(t)}{dt}, \frac{dv_{yi}(t)}{dt^2}, \frac{dv_{zi}(t)}{dt}\right], \\ &= \frac{1}{m_i}\left(-\rho_i \vec{v}(t) + \vec{f}^s_i(t) + \vec{f}^e_i(t)\right). \end{aligned}$$

– Solve using your favourite ODE solver.
  The simplest technique is the Euler step:
  * Pick a Δt.
  * Then compute the values of all positions at $t + \Delta t$ from the positions at t by discretizing the differential equations:

$$\begin{aligned}
\vec{a}_i(t + \Delta t) &\leftarrow \frac{\Delta t}{m_i}\left(-\rho_i \vec{v}(t) + \vec{f}_i^{\,s}(t) + \vec{f}_i^{\,e}(t)\right), \\
\vec{v}_i(t + \Delta t) &\leftarrow v_i(t) + \vec{a}(t)\Delta t, \\
P_i(t + \Delta t) &\leftarrow P_i(t) + \vec{v}(t)\Delta t.
\end{aligned}$$

29.12 Human Motion

Walking

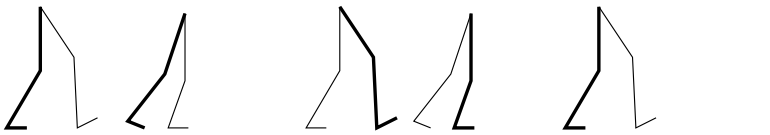
Modeling walking is hard

- Human motion (walking) is extremely complex
  Fine grained balancing act
- Human walking involves more than just the feet and legs
  Hips, body, shoulder, head
- Human perception of walking is very precise
  Poor walking immediately spotted
- Task is still an area of active research

We will look at some basics

Walk Cycle

- Walking is mostly a cycle of motions
  Have to adapt each time through cycle for terrain, etc.
- In walking, at least one foot on ground
- Left stance, right swing, right stance, left swing
  Repeat



- Hip movement: rotation
- Hip rotation requires knee flexion
- Ankle, toe joints
- Can animate leg motion by specifying angles of all joints as function of time

29.13 Sensor-Actuator Networks

Sensor-Actuator Networks

- van de Panne
- Model creature as set of
  - Links
  - Sensors
  - Actuators
- SAN relates sensors and actuators
- Control uses sensor data to apply forces thru actuators

Links

- Rigid
- Have mass
- May restrict to planar for simplicity

Sensors : 4 types

- Touch (picture)
- Angle (picture)
- Eye - used to find target
- Length (picture)

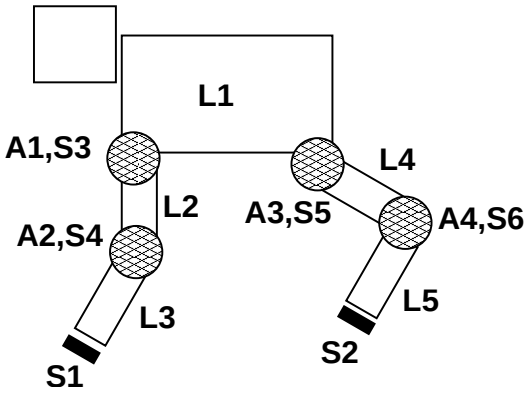
Actuators

- Length/linear
  - push/pull points
  - max/min, strength
- Angle
  - create torques
  - max/min angle, strength

Motion

- Evaluation metric
  - Distance traveled
  - Don't fall over
- Generate random controllers and evaluate with metric
- Try lots, and fine tune the best

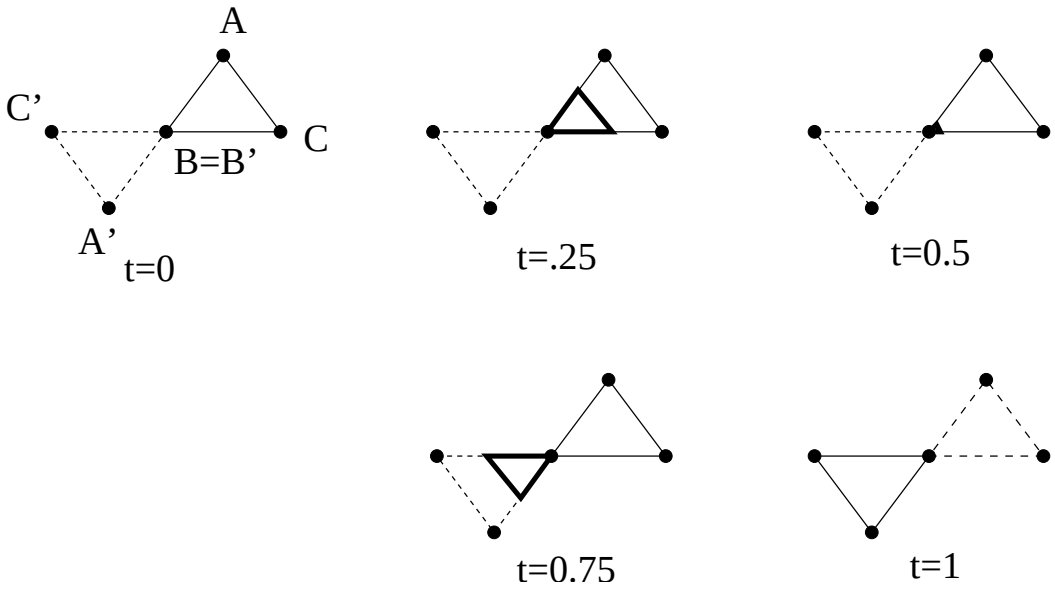
SAN Example



29.14 Morphing

- Morphing is lerping between images.
- Nice effects at low cost.
- Key issues:
  - Partitioning images into pieces that correspond
    E.g., eyes to eyes, etc.
  - Lerping pieces (geometry)
  - Lerping pieces (pixels, colours)
  - Filtering
- Developed at NRC in the 1970's (line drawings).
- Make it quick, don't look too closely at intermediate images.

- Simplest morph linearly interpolates each point
  Mostly okay, but can cause problems:



  Often tweak by hand to get rid of such problems

- More sophisticated morphs introduce other problems
- Desired properties:
  Vertex trajectories smooth and continuous
  Triangles don't cross each other
  Minimize triangle size variations
  Linear when other properites not violated

29.15 Motion Capture

- Keyframing complex actions (walking, etc) requires a lot of key frames.
- Instead, capture motion from real objects.
  Creatures and/or actors performing the desired motion.
- The positions of key points on the actor are tracked
  these motions are mapped to corresponding points in the model.
- May be necessary to generate secondary actions for the computer model.
  (tracking is expensive)
- Tracking technologies include

  1. Electromagnetic position and orientation sensors.

   - Requires a wire to each sensor.
   - Readings can be distorted by metal and magnetic fields.
   - Limited working volume.
2. Ultrasonic rangefinder triangulation.
   - Cheaper but less accurate than magnetic sensors.
3. Optical triangulation.
   - Reflective or emissive markers are attached to objects
   - Two cameras triangulate position
   - Can track more points cheaply, without wires
   - Points can be occluded.
4. Body suit.
   - Fibers in instrumented clothing detect angles of all joints.

5. Rotoscoping.
   - Arbitrary video from two cameras converted to motion paths
     * manual selection of key points
     * computer vision
   - Prone to error and occlusion.
   - Tedious if done manually.
6. Puppetry.
   - Real-time data collected from an arbitrary input device
   - Mapped to an computer model, which is displayed in real time during capture.
   - A special-purpose input device in the shape of the object being animated may be built.

- Motion capture tends to require more-or-less exotic technology.
- Even puppetry requires at least real-time graphics.
- Problems with motion capture include:
  - How do we map motion paths to objects of a different shape and scale (the ostrich-to-T. Rex problem)?
  - How do we deal with noise and missing data?
  - How can we reduce the number of samples?
  - How can we generalize motion? I.e. we have a walk and a turn. How do we get an actor to follow a path?

29.16 Flocking

- Want to animate groups of animals
  Birds, fish, herds
- Motion of individuals is semi-independent
  Too many individuals to key frame each one
- Fewer elements than particle systems, but more interelement interaction
- Relative placing not rigid
  Flock of birds vs airplanes in formation
  (migrating geese, etc., being more like the latter)

Two main forces at work

- Collision avoidance
  - Avoid hitting objects (trees, buildings, etc.)
  - Avoid hitting one another
- Flock centering
  - Each member trying to remain a member of flock
  - Global flock centering too restricting
    Does not allow flock splitting to pass around objects

Local Control

- Controlling each flock member with local behaviour rules
  Computationally desirable
  Appears to be how flocks work in real world
- No reference to global conditions of flock, environment
- Three things to model:
  - Physics
    Gravity, collisions
  - Perception
    Information flock member has with which to make decisions
  - Reasoning/reaction
    Rules by which flock member decides where to go

Perception

- Aware of itself and two or three neighbors
- What's in front of it
- Doesn't follow a designated leader
- No knowledge of global center

Reasoning/reaction

- Collision avoidance
- Flock centering
- Velocity matching

  Try to match velocity of neighbors

  Helps with flock centering and helps avoid collisions

Additional details

- Global control

  Animator needs to direct flock.

- Flock leader

  Usually one member whose path is scripted (global control)

  Not realistic (leader changes in real flocks),
  but usually not noticed and easier to deal with

- Splitting and Rejoining

  Flocks pass around objects, splitting them

  Difficult to balance rejoining with collision avoidance

30 Assignments

30.1 Assignment 0: Introduction

The goals of this assignment include:

- Familiarization with the course computing environment.
- Setting up your account and directories.
- Modifying a UI.
- A trial assignment submission.

This assignment is not worth any marks. However, we *strongly suggest you do it*:

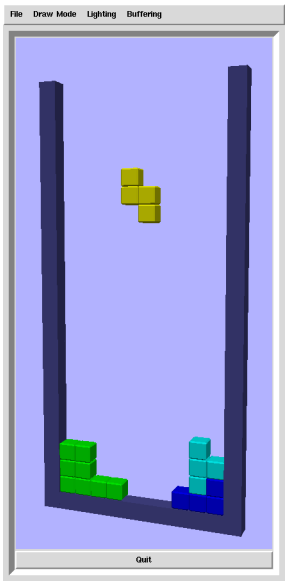
- It's easy.
- You have to learn most of it eventually anyways.
- It'll save you from losing marks over dumb mistakes in protocol.

30.2 Assignment 1: Introduction to OpenGL

In this assignment you will learn how to use OpenGL to render a polygonal model.

- Draw polygons
- Lighting
- Transformations
- UI

Screen Shots



Sample screen shots. Some details may vary term to term.

30.3 Assignment 2: Frames and Perspective

This assignment includes:

- Modeling, Viewing, and Perspective Transformations
- 3D Line/Plane Clipping
- Menus and Valuators

Your program will manipulate a cube and 3 sets of coordinate frames.

The cube is in a modelling frame, is transformed to a world frame, and then into a viewing frame which is projected onto a window.

You will draw gnomons for the modelling and world frames.

Frames

There are six entities you need to consider:

The Cube Model: Defined on the unit points $[\pm 1, \pm 1, \pm 1]$.

The Cube Model Frame: For entering the initial coordinates of the Cube Model.

The Cube Model Gnomon: A *graphic representation* of the Cube Model Frame.

The World Frame: A coordinate system for describing the position and orientation of objects in the scene.

The World Frame Gnomon: A *graphic representation* of the World Frame.

The Viewing Frame: A *coordinate system* for representing a view of the scene relative to the eyepoint and window.

Features

Your program will support the following features:

Menus: Selection of transformation mode and coordinate system.

Valuators: Mouse movement to specify scalar parameters.

Modelling Transformations: Rotation, translation, scale.

Viewing Transformations: Camera rotation and translation, perspective.

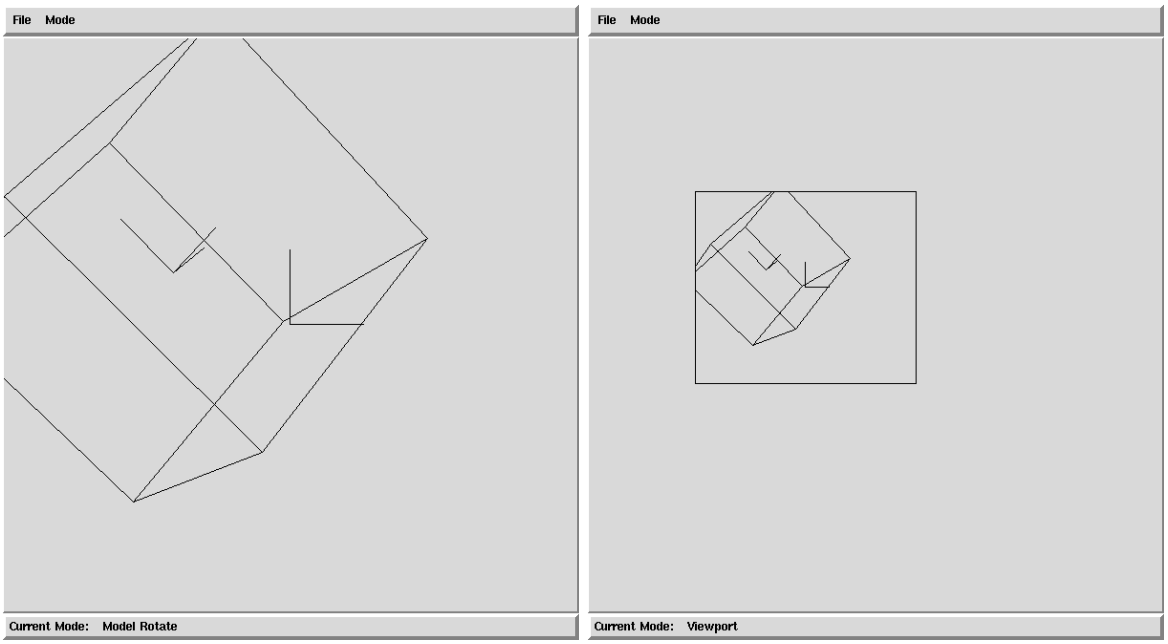
3D Line Clipping: To avoid divide by zero, you need to clip all lines to the near plane.

Viewport: Clip the drawing to a 2D viewport.

OpenGL may be used for only 2D line drawing.

You must implement all transformations and clipping yourselves.

Screen Shots



Sample screen shots. Some details may vary term to term.

30.4 Assignment 3: Hierarchical Modelling

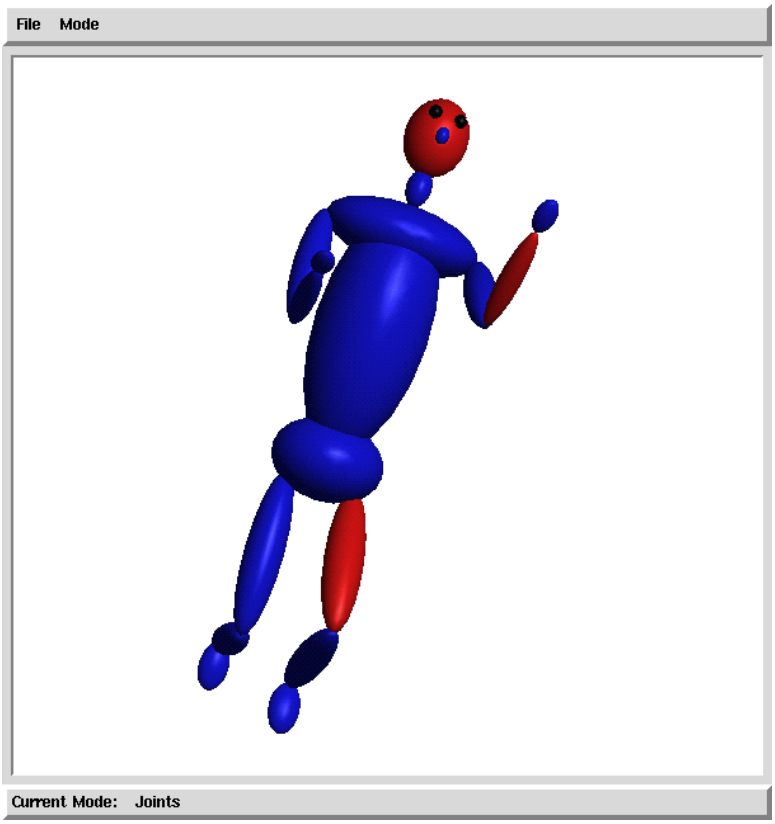
This assignment includes:

- Hierarchical Models (and data structures).
- Matrix Stacks
- 3D Picking
- Z-Buffer Hidden Surface Removal
- Lighting Models and Shaded Polygons
- Display Lists
- Virtual Sphere or Arcball

You may use any feature of OpenGL you desire, including transformations and lighting models, to implement this assignment.

Lua should be used as the modelling language.

Screen Shots



Sample screen shots. Some details may vary term to term.

30.5 Assignment 4: A Raytracer

- In this assignment you will investigate the implementation of
  - Ray Tracing
  - Colour and Shading
- Some sample code will be provided to get you started.
- Your raytracer will implement at least spheres, cubes, polygons, and the Phong lighting model.
- Only primary and shadow rays are required in the basic implementation.

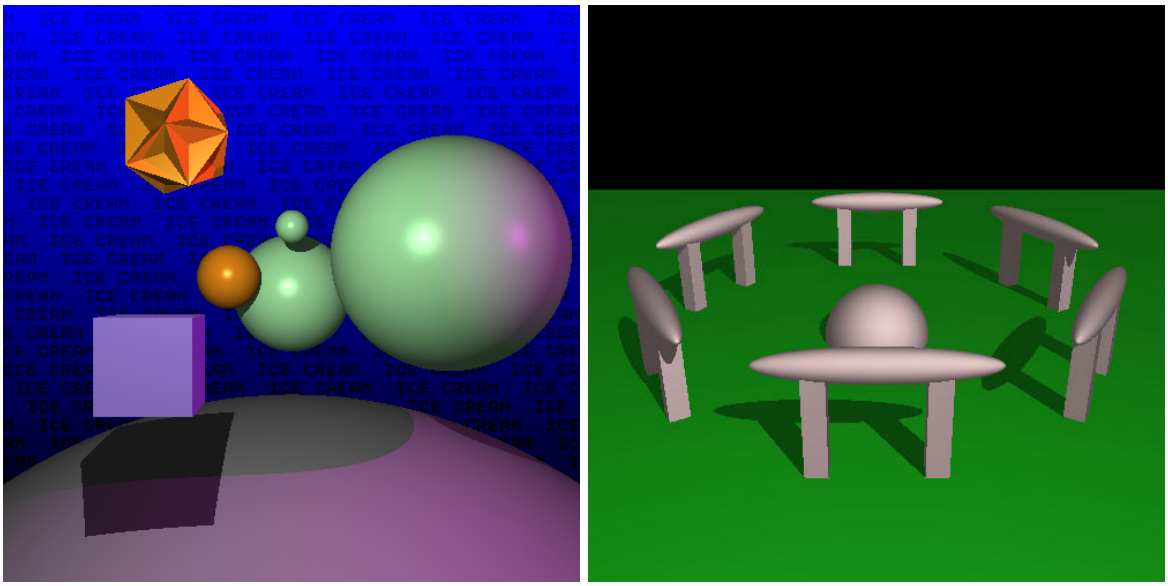
Additional Features

You will also be asked to implement one additional feature of your choice. For example, you could implement:

- Secondary rays for reflective surfaces
- Depth of field using (jittered) aperture supersampling

- Antialiasing using (jittered) pixel supersampling
- A fisheye lens
- Texture or bump mapping
- CSG operations
- Additional object types

Screen Shots



Sample screen shots. Some details may vary term to term.

30.6 Assignment 5: The Project

- The project submission is distributed over three phases:
  - A proposal
  - A revised proposal, addressing our comments.
  - A final submission and demonstration.
- Your project should have a significant computer graphics content, but is otherwise unconstrained.
- A project does not have to have an interactive user interface, but if not you will have to work harder on the algorithms
- You will have to prepare documentation for your project, and the quality of this documentation will strongly influence your mark.
- Ask us for project suggestions; also, it would be wise to run your idea past us *before* submitting your proposal.

Objective List

- Your proposal needs to include an Objectives list, which we will use to mark you.
- No objectives from assignments!
- Roughly...
  - 1/3 to 1/2 easy objectives
    Modelling, texture mapping, alpha blending
  - A few medium ones
    L-systems, reflection, 2D collisions
  - 1 or 2 hard ones
    3D collisions, hardware tricks, stuff from papers
- Try to break hard objectives into multiple pieces

Common Objectives

- Modelling/scene
- Animation
- User interface (more complex interaction)
- Texture mapping
- Artifical Intelligence, sound
  If your other objectives had enough graphics content
- Ray tracer:
  - Must tell us your A4 extra feature
  - CSG
  - Reflection, refraction, anti-aliasing
  - Extra primitives (torus)

Warm/cold Fuzzies About Objectives

- Think about what subjective mark you could get
  Technical, artistic, documentation, difficulty
- Negative subjective marks for not using what you've learned
  eg, flat shading in OpenGL projects
- Get advise from TAs and profs
- Beware of too easy/too hard projects
  You can always do extra stuff

- Make objectives, not subjective
  "Good UI" is subjective...
- When wondering if something makes a good project, try making an objective list for it.

OpenGL vs Ray Tracing

OpenGL Projects

- Interactive
- Avoid UI intensive projects
- Try to have a purpose (game, etc)

Ray Tracing

- Must get 9/10 on A4 to do ray tracing project
- Easier to come up with objective list
- **Design objectives around a scene**
  Photograph or real object you want to ray trace
- Nice scene must be one objective!
  Describe it

Other Stuff

- Technical outline should clarify objectives
- You can start before you get proposal back
- Avoid dynamic memory
- See TAs if you have problems
  Don't spend 2 days hacking at it
- If running out of time, prof or TA can give advice on what objectives to tackle
- **Project is marked on objective list of revised proposal.**